APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS

MODULE III

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.

Reference:

Elmasri R. and S. Navathe, Database Systems: Models, Languages, Design and Application Programming, Pearson Education, 2013.

SQL DML

Basic Retrieval Queries in SQL

SQL has one basic statement for retrieving information from a database: the **SELECT** statement. SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values. Hence, in general, an **SQL** table is not a *set of tuples*, because a set does not allow two identical members; rather, it is a **multiset** (sometimes called a *bag*) of tuples.

The SELECT-FROM-WHERE Structure of Basic SQL Queries

The basic form of the SELECT statement, sometimes called a **mapping** or a **select-from-where block**, is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

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

where

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

In SQL, the basic logical comparison operators for comparing attribute values with one another and with literal constants are =, <, <=, >, >=, and <>. These correspond to the relational algebra operators =, <, ≤, >, ≥, and ≠, respectively.

Query 0. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

Q0: **SELECT** Bdate, Address
FROM EMPLOYEE
WHERE Fname='John' **AND** Minit='B' **AND** Lname='Smith';

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

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

In the WHERE clause of Q1, the condition Dname = 'Research' is a **selection condition** that chooses the particular tuple of interest in the DEPARTMENT table, because Dname is an attribute of DEPARTMENT. The condition Dnumber = Dno is called a **join condition**, because it combines two tuples: one from DEPARTMENT and one from EMPLOYEE, whenever the value of Dnumber in DEPARTMENT is equal to the value of Dno in EMPLOYEE.

A query that involves only selection and join conditions plus projection attributes is known as a **select-project-join** query. The next example is a select-project-join query with *two* join conditions.

Query 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.

Q2: **SELECT** Pnumber, Dnum, Lname, Address, Bdate
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE Dnum=Dnumber **AND** Mgr_ssn=Ssn **AND** Plocation='Stafford';

The join condition Dnum = Dnumber relates a project tuple to its controlling department tuple, whereas the join condition Mgr_ssn = Ssn relates the controlling department tuple to the employee tuple who manages that department.

Ambiguous Attribute Names, Aliasing, Renaming, and Tuple Variables

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in *different relations.* If this is the case, and 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. To illustrate this, suppose that the Dno and Lname attributes of the EMPLOYEE relation were called Dnumber and Name, and the Dname attribute of DEPARTMENT was also called Name; then, to prevent ambiguity, query Q1 would be rephrased as shown in Q1A.

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

We can also create an *alias* for each table name to avoid repeated typing of long table names (see Q8 below).

Query 8. For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

Q8: **SELECT** E.Fname, E.Lname, S.Fname, S.Lname
FROM EMPLOYEE **AS** E, EMPLOYEE **AS** S
WHERE E.Super_ssn=S.Ssn;

In this case, we are required to declare alternative relation names E and S, called **aliases** or **tuple variables**, for the EMPLOYEE relation. An alias can follow the keyword **AS**, as shown in Q8, or it can directly follow the relation name—for example, by writing EMPLOYEE E, EMPLOYEE S in the FROM clause of Q8.

Unspecified WHERE Clause and Use of the Asterisk

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

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

```
Q9:  SELECT Ssn
     FROM EMPLOYEE;
Q10: SELECT Ssn, Dname
     FROM EMPLOYEE, DEPARTMENT;
```

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.*

For example, query Q1C retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5, query 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, and Q10A specifies the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT relations.

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

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

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.

An SQL table with a key is restricted to being a set, since the key value must be distinct in each tuple. If we *do want* to eliminate duplicate tuples from the result of an SQL query, we use the keyword **DISTINCT** in the SELECT clause, meaning that only distinct tuples should remain in the result. In general, a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL does not.

Specifying SELECT with neither ALL nor DISTINCT—as in our previous examples— is equivalent to SELECT ALL.

Q11 retrieves the salary of every employee; if several employees have the same salary, that salary value will appear as many times in the result of the query

Query 11. Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

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

Q11A: SELECT DISTINCT Salary
FROM EMPLOYEE;

SQL has directly incorporated some of the set operations from mathematical *set theory*, which are also part of relational algebra. There are 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.

Query 4. 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.

SQL also has corresponding multiset operations, which are followed by the keyword **ALL** (UNION ALL, EXCEPT ALL, INTERSECT ALL). Their results are multisets (duplicates are not eliminated).

Substring Pattern Matching and Arithmetic Operators

The first feature allows comparison conditions on only parts of a character string, using the **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.

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

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

To retrieve all employees who were born during the 1950s, we can use Query Q12A. 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.

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

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

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.

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (–), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains.

For example, 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.

Query 13. 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';

Another comparison operator, which can be used for convenience, is **BETWEEN**, which is illustrated in Query 14.

Query 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;

Ordering of Query Results

SQL allows the user to order the tuples in the result of a query by the values of one or more of the attributes that appear in the query result, by using the **ORDER BY** clause.

Query 15. Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, then first name.

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

The default order is in ascending order of values. We can specify the keyword **DESC** if we want to see the result in a descending order of values. The keyword **ASC** can be used to specify ascending order explicitly. For example, 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**

More Complex SQL Retrieval Queries

Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. NULL is used to represent a missing value, but that it usually has one of three different interpretations—value *unknown* (exists but is not known), value *not available* (exists but is purposely withheld), or value *not applicable* (the attribute is undefined for this tuple).

1. **Unknown value.** A person's date of birth is not known, so it is represented by NULL in the database.
2. **Unavailable or withheld value.** A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.
3. **Not applicable attribute.** An attribute LastCollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person.

SQL allows queries that check whether an attribute value is **NULL**. Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operators **IS** or **IS NOT**.

Query 18. Retrieve the names of all employees who do not have supervisors.

```
Q18: SELECT Fname, Lname
     FROM EMPLOYEE
     WHERE Super_ssn IS NULL;
```

Nested Queries, Tuples, and Set/Multiset Comparisons

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries**, which are complete select-from-where blocks within the WHERE clause of another query. That other query is called the **outer query**.

Q4A introduces the comparison operator **IN**, which compares a value *v* with a set (or multiset) of values *V* and evaluates to **TRUE** if *v* is one of the elements in *V*.

The first nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as manager, while the second nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as worker. In the outer query, we use the **OR** logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

Q4A:
```
SELECT DISTINCT Pnumber
FROM PROJECT
WHERE Pnumber IN
(SELECT Pnumber
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE Dnum=Dnumber AND Mgr_ssn=Ssn AND Lname='Smith')
OR
Pnumber IN
(SELECT Pno
FROM WORKS_ON, EMPLOYEE
WHERE Essn=Ssn AND Lname='Smith');
```

SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

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

This query will 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. 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.

Correlated Nested Queries

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

For *each* EMPLOYEE tuple, evaluate the nested query, which retrieves the Essn values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the Ssn value of the EMPLOYEE tuple is *in* the result of the nested query, then select that EMPLOYEE tuple.

```
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;
```

The EXISTS and UNIQUE Functions in SQL

The 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.

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

```
Q16B: 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 AND E.Fname =
      D.Dependent_name);
```

For each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same Essn, Sex, and Dependent_name as the EMPLOYEE tuple; if at least one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple. In general, 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.

Query 6. Retrieve the names of employees who have no dependents.

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

Explicit Sets and Renaming of Attributes in SQL

It is also possible to use an **explicit set of values** in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses in SQL.

Query 17. Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.

```
Q17: SELECT DISTINCT Essn
     FROM WORKS_ON
     WHERE Pno IN (1, 2, 3);
```

Joined Tables in SQL and Outer Joins

The concept of a **joined table** (or **joined relation**) was incorporated into SQL to permit users to specify a table resulting from a join operation *in the* FROM *clause* of a query.

For example, consider query Q1, which retrieves the name and address of every employee who works for the 'Research' department.

```
Q1A: SELECT Fname, Lname, Address
     FROM (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
     WHERE Dname='Research';
```

It is also possible to *nest* join specifications; that is, one of the tables in a join may itself be a joined table. This allows the specification of the join of three or more tables as a single joined table, which is called a **multiway join**.

Q2A: **SELECT** Pnumber, Dnum, Lname, Address, Bdate
FROM ((PROJECT **JOIN** DEPARTMENT **ON** Dnum=Dnumber)
JOIN EMPLOYEE **ON** Mgr_ssn=Ssn)
WHERE Plocation='Stafford';

Aggregate Functions in SQL

Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary. **Grouping** is used to create subgroups of tuples before summarization.

A number of built-in aggregate functions exist: **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**.

The 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

Query 19. 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;

Query 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';

Queries 21 and 22. Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

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

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.

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

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

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*.

SQL has a **GROUP BY** clause for this purpose. 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 attribute(s).

Query 24. 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;

In Q24, the EMPLOYEE tuples are partitioned into groups—each group having the same value for the grouping attribute Dno.

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;
```

Sometimes we want to retrieve the values of these functions only for *groups that satisfy certain conditions*. For example, suppose that we want to modify Query 25 so that only projects with more than two employees appear in the result. SQL provides a **HAVING** clause, which can appear in conjunction with a GROUP BY clause, for this purpose. 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.

Query 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;
```

Specifying Constraints as Assertions and Actions as Triggers

Specifying General Constraints as Assertions in SQL

In SQL, users can specify general constraints via **declarative assertions**, using the **CREATE ASSERTION** statement of the DDL. Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.

For example, to specify the constraint that *the salary of an employee must not be greater than the salary of the manager of the department that the employee works for* in SQL, we can write the following assertion:

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

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

Introduction to Triggers in SQL

Another important statement in SQL is CREATE TRIGGER. In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied.

For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation. A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs. The CREATE TRIGGER statement is used to implement such actions in SQL.

Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database.

Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external stored procedure SALARY_VIOLATION, which will notify the supervisor.

R5: **CREATE TRIGGER** SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN **ON** EMPLOYEE
FOR EACH ROW
WHEN (**NEW**.SALARY > (**SELECT** SALARY **FROM** EMPLOYEE
WHERE SSN = **NEW**.SUPERVISOR_SSN))
INFORM_SUPERVISOR(**NEW**.Supervisor_ssn, **NEW**.Ssn);

The trigger is given the name SALARY_VIOLATION, which can be used to remove or deactivate the trigger later.

A typical trigger has three components:

1. The **event(s)**: These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. The person who writes the trigger must make sure that all possible events are accounted for. In some cases, it may be necessary to write more than one trigger to cover all possible cases. These events are specified after the keyword **BEFORE** in our example, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.
2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated. If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed. The condition is specified in the WHEN clause of the trigger.
3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed. In this example, the action is to execute the stored procedure INFORM_SUPERVISOR.

Views (Virtual Tables) in SQL

Concept of a View 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.

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.

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;
```

In V1, we did not specify any new attribute names for the view WORKS_ON1 (although we could have); in this case,WORKS_ON1 *inherits* the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON.

View V2 explicitly specifies new attribute names for the view DEPT_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.

We can now specify SQL queries on a view—or virtual table—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';
```

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

The problem of efficiently implementing a view for querying is complex. Two main approaches have been suggested. One strategy, called **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.

The second strategy, called **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. In this case, 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.

- 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.

It is also possible to define a view table in the **FROM clause** of an SQL query. This is known as an **in-line view**. In this case, the view is defined within the query itself.

PHYSICAL DATA ORGANIZATION

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**.

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**.

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.

Files of Unordered Records

Records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file. Such an organization is called a heap or pile file.

Inserting a new record is very efficient. The last disk block of the file is copied into a buffer, the new record is added, and the block is then rewritten back to disk. The address of the last file block is kept in the file header. However, searching for a record using any search condition involves a linear search through the file block by block—an expensive procedure. To delete a record, a program must first find its block, copy the block into a buffer, delete the record from the buffer, and finally rewrite the block back to the disk. This leaves unused space in the disk block.

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.

Indexes can be characterized as dense or sparse

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

Types of Single-Level Indexes

Primary Index

- Defined on an ordered data file
- The data file is ordered on a *key field*
- Includes one index entry *for each block* in the data file; the index entry has the key field value for the *first record* in the block, which is called the *block anchor*
- A similar scheme can use the *last record* in a block.
- 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.

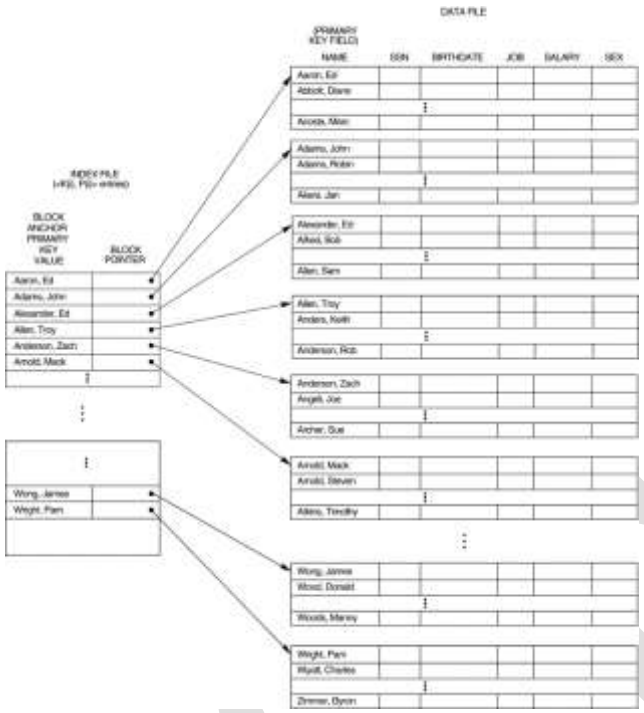


Figure 5.1: Primary Index on the Ordering Key Field of the File

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 = $\lfloor B / R \rfloor$ = $\lfloor 1024 / (9 + 6) \rfloor$ = 68 entries/block

Total no. of index entries = No. of blocks in data file = 3000

No. of blocks needed = $\lceil r/bfr \rceil$ = $\lceil 3000/68 \rceil$ = 45 blocks

If binary search = log2(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.

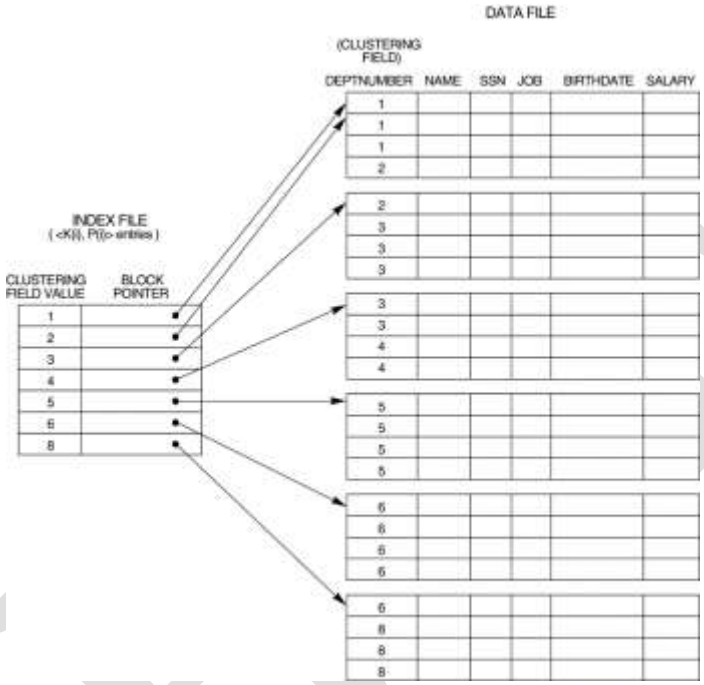


Figure 5.2: A clustering index on the DEPTNUMBER ordering nonkey field of an EMPLOYEE file.

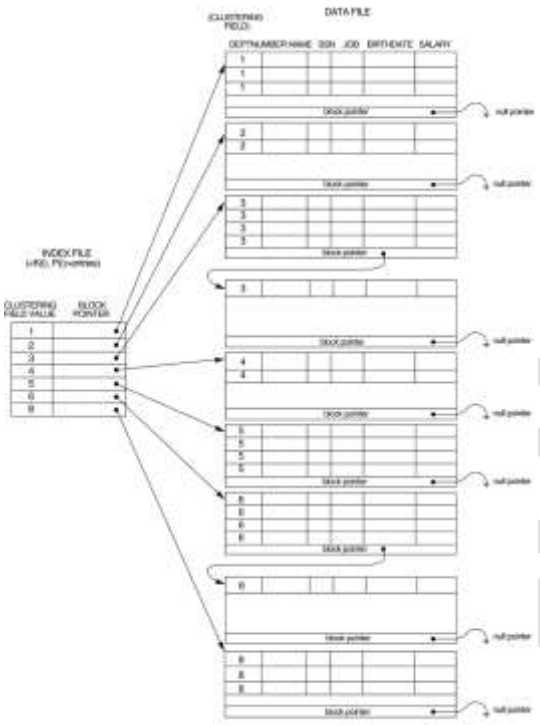


Figure 5.3: Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

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 nonkey with duplicate values.
- The index is an ordered file with two fields.
- The first field is of the same data type as some *nonordering field* of the data file 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*

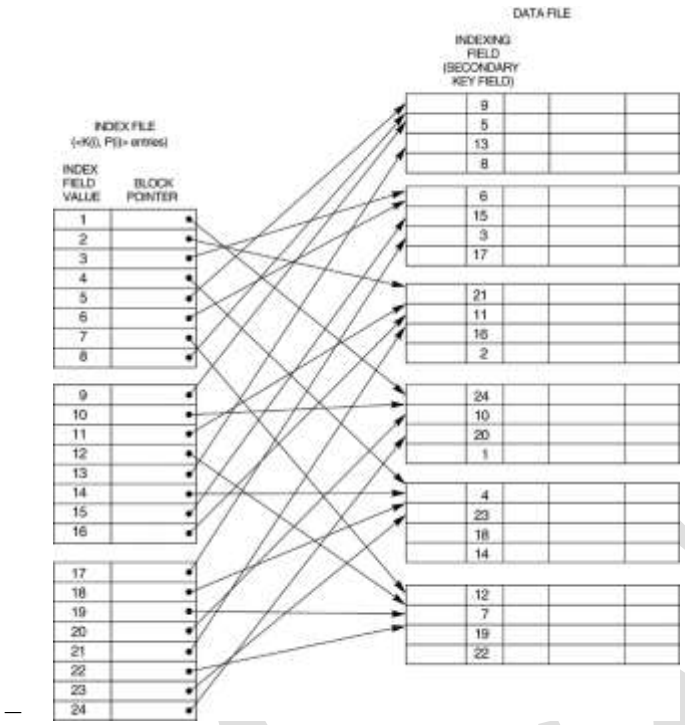


Figure 5.4: A dense secondary index (with block pointers) on a nonordering key field of a file.

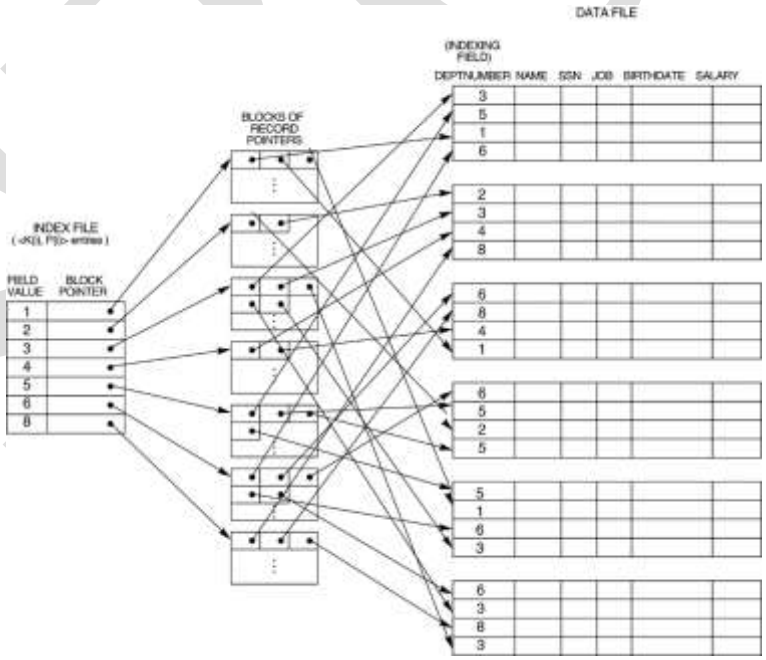


Figure 5.5: A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

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

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

Multi-Level Indexes

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

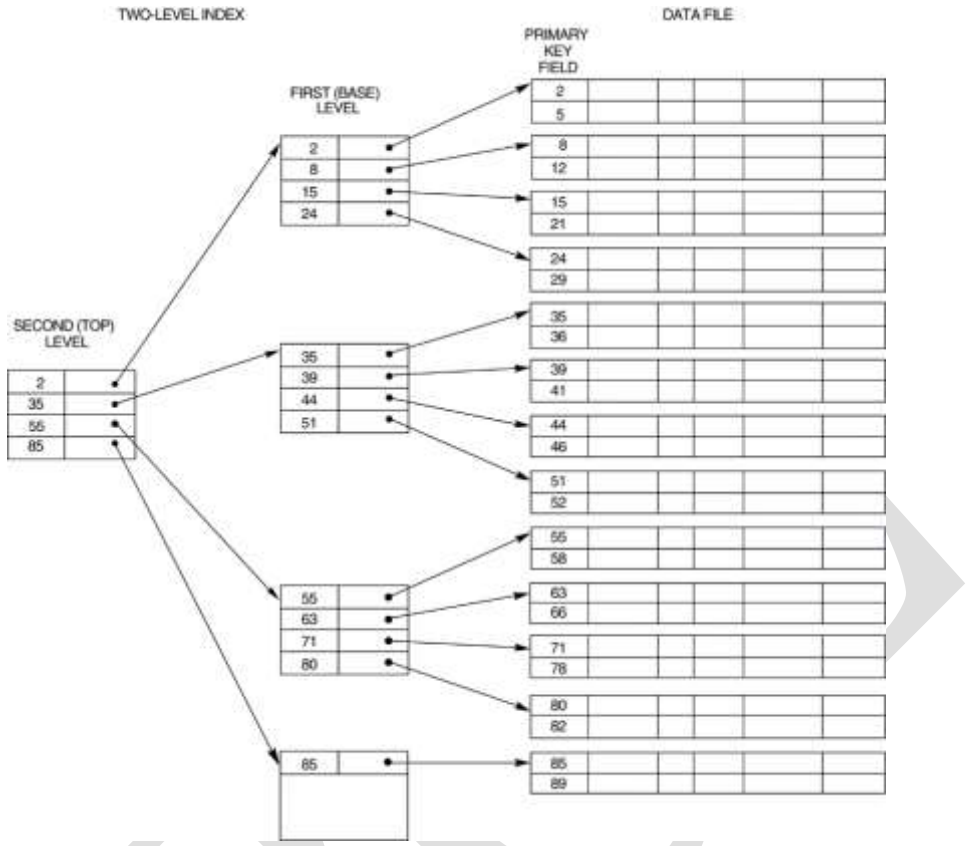


Figure 5.6: A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.

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

- Because of the insertion and deletion problem, most multi-level indexes use B-tree or B+-tree data structures, which leave 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

Difference between B-Tree and B+-Tree

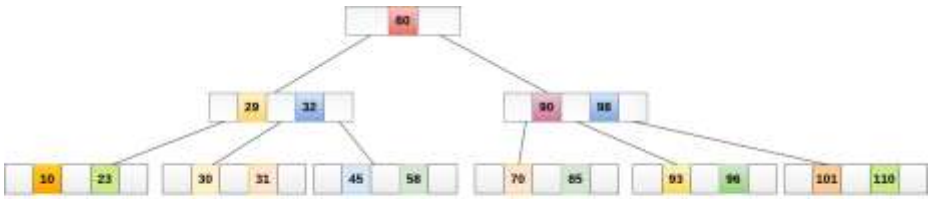
- In a B-tree, pointers to data records exist at all levels of the tree
- In a B+-tree, all pointers to data records exists at the leaf-level nodes
- A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree

B – Trees

- The **B-tree** is the classic disk-based data structure for indexing records based on an ordered key set.
- A B-Tree of order m can have at most m-1 keys and m children.
- One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

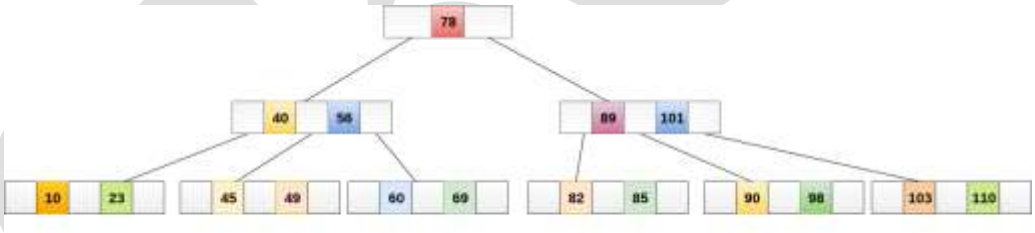
Properties

- Every node in a B-Tree contains at most m children.
- Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
- The root nodes must have at least 2 nodes.
- All leaf nodes must be at the same level.
- It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.



Searching in a B – Tree

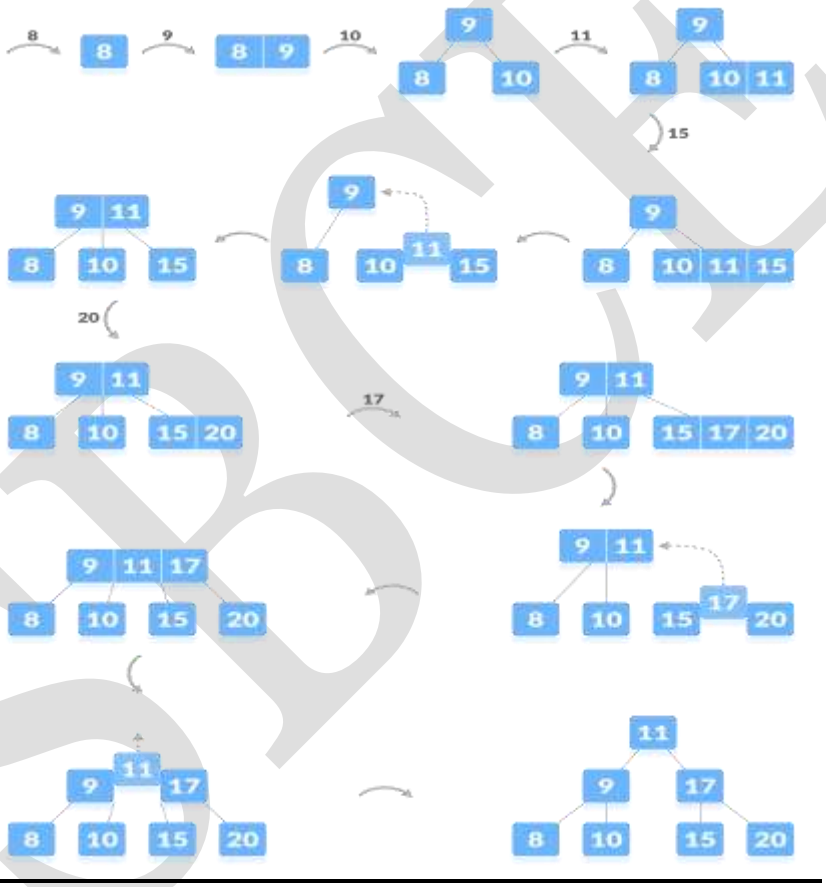
- Searching in B Trees is similar to that in Binary search tree.
- For example, if we search for an item 49 in the following B Tree.
- The process will something like following :
- Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
- Since, 40<49<56, traverse right sub-tree of 40.
- 49>45, move to right. Compare 49.
- Match found, return.



Insertion in a B-Tree

- Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.
- Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.

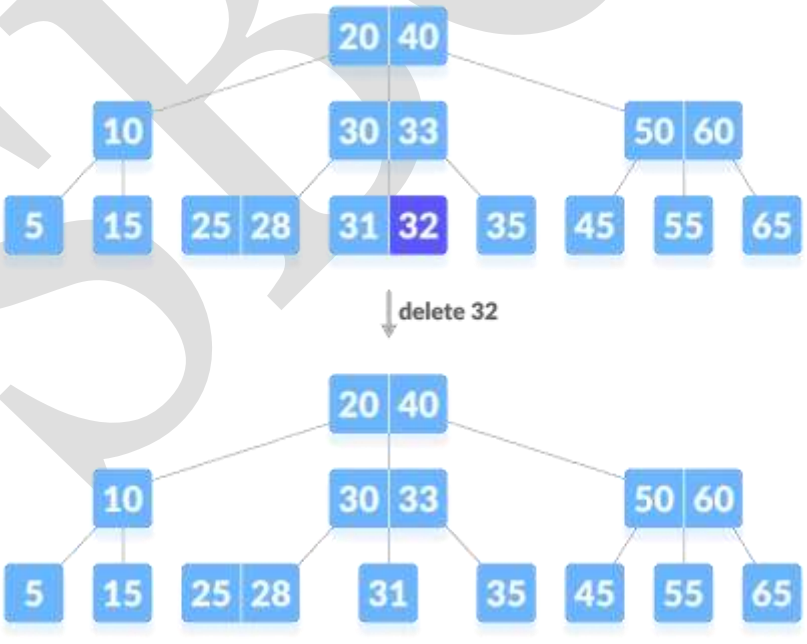
- If the leaf node contain less than m-1 keys then insert the element in the increasing order.
- Else, if the leaf node contains m-1 keys, then follow the following steps.
  - Insert the new element in the increasing order of elements.
  - Split the node into the two nodes at the median.
  - Push the median element upto its parent node.
  - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

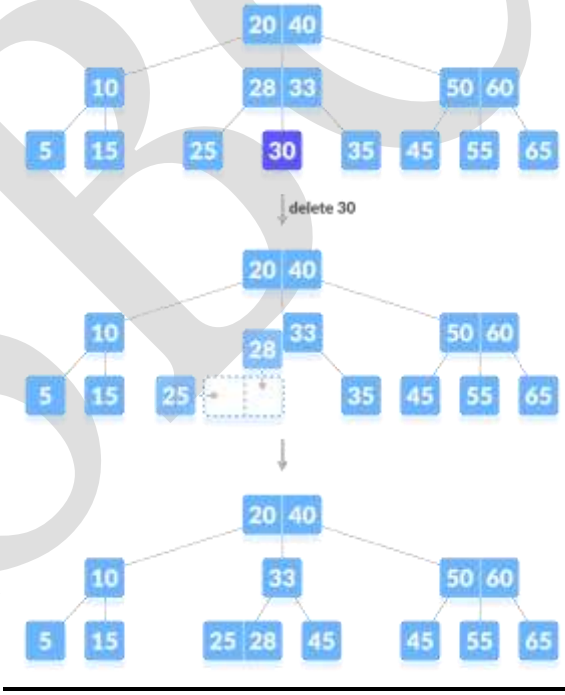


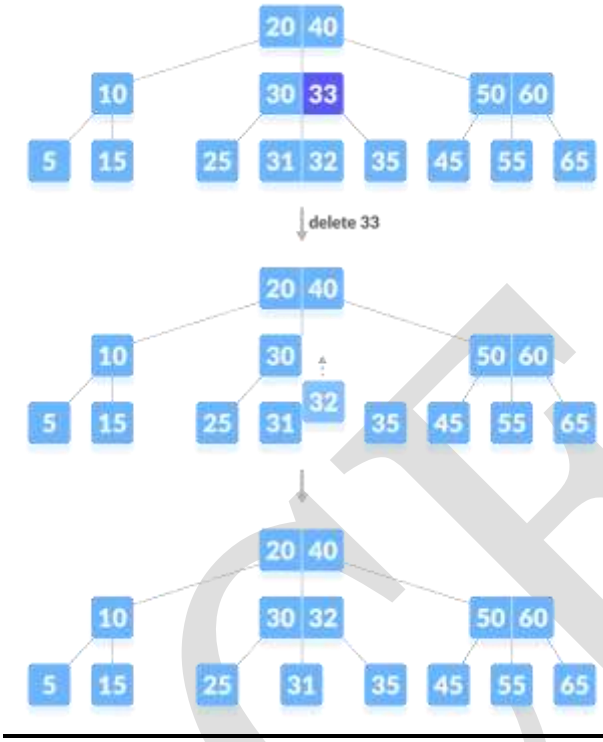
Deletion in a B-Tree

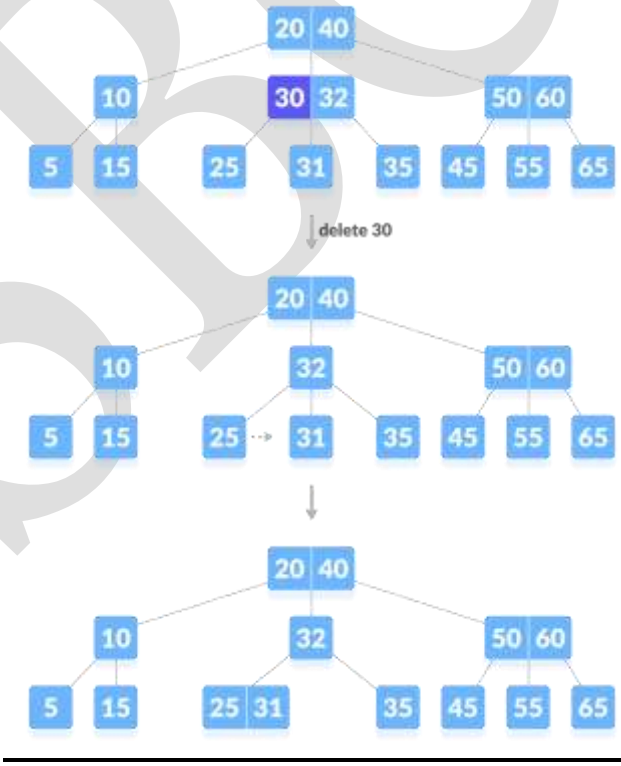
- Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.
- Locate the leaf node.

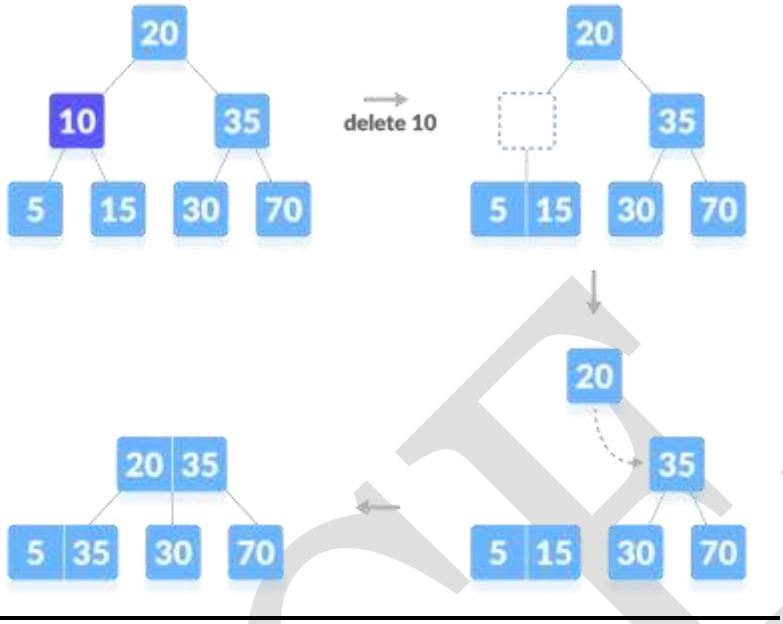
- If there are more than m/2 keys in the leaf node then delete the desired key from the node.
- If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
    - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
- If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
- If parent is left with less than m/2 nodes then, apply the above process on the parent too.
- If the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.









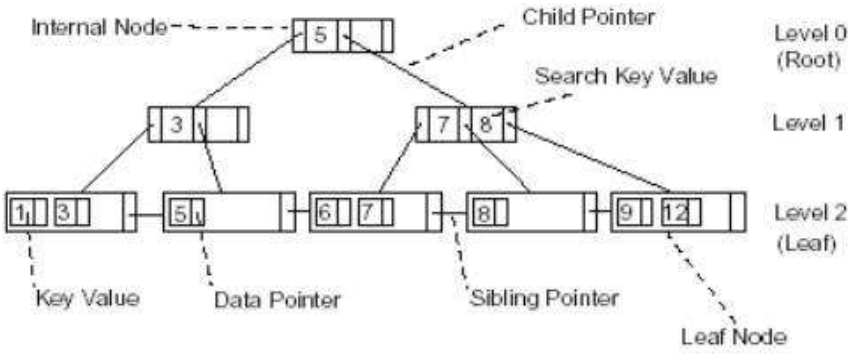


B$^+$-Tree

The **B-tree** is the classic disk-based data structure for indexing records based on an ordered key set. The **B$^+$-tree** (sometimes written B+-tree, B+tree, or just B-tree) is a variant of the original B-tree in which all records are stored in the leaves and all leaves are linked sequentially. The B+-tree is used as a (dynamic) indexing method in relational database management systems.

B+-tree considers all the keys in nodes except the leaves as dummies. All keys are duplicated in the leaves. This has the advantage that is all the leaves are linked together sequentially, the entire tree may be scanned without visiting the higher nodes at all.

B+-Tree Structure



- A B + Tree consists of one or more blocks of data, called *node*s, linked together by pointers. The B + -Tree is a tree structure. The tree has a single node at the top, called the *root nod*e. The root node points to two or more blocks , called *child node*s. Each child nodes points to further child nodes and so on.
- The B + -Tree consists of two types of (1) *internal nodes* and (2) *leaf node*s:
- Internal nodes point to other nodes in the tree.
- Leaf nodes point to data in the database using *data pointer*s. Leaf nodes also contain an additional pointer, called the *sibling pointe*r, which is used to improve the efficiency of certain types of search.
- All the nodes in a B + -Tree must be at least half full except the root node which may contain a minimum of two entries. The algorithms that allow data to be inserted into and deleted from a B + -Tree guarantee that each node in the tree will be at least half full.
- Searching for a value in the B + -Tree always starts at the root node and moves downwards until it reaches a leaf node.
- Both internal and leaf nodes contain *key values* that are used to guide the search for entries in the index.
- The B + -Tree is called a *balanced tree* because every path from the root node to a leaf node is the same length. A balanced tree means that all searches for individual values require the same number of nodes to be read from the disc.

Internal Nodes

- An *internal node* in a B + -Tree consists of a set of *key values* and *pointer*s. The set of keys and values are ordered so that a pointer is followed by a key value. The last key value is followed by one pointer.
- Each pointer points to nodes containing values that are *less than or equal to* the value of the key immediately to its right
- The last pointer in an internal node is called the *infinity pointe*r. The infinity pointer points to a node containing key values that are greater than the last key value in the node.
- When an internal node is searched for a key value, the search begins at the leftmost key value and moves rightwards along the keys.
- If the key value is less than the sought key then the pointer to the left of the key is known to point to a node containing keys less than the sought key.
- If the key value is greater than or equal to the sought key then the pointer to the left of the key is known to point to a node containing keys between the previous key value and the current key value.

Leaf Nodes

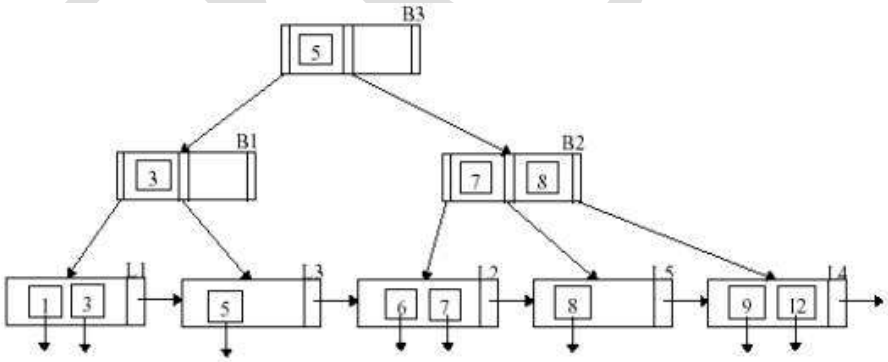
- A *leaf node* in a B + -Tree consists of a set of *key values* and *data pointer*s. Each key value has one data pointer. The key values and data pointers are ordered by the key values.
- The data pointer points to a record or block in the database that contains the record identified by the key value. For instance, in the example, above, the pointer attached to key value 7 points to the record identified by the value 7.
- Searching a leaf node for a key value begins at the leftmost value and moves rightwards until a matching key is found.
- The leaf node also has a pointer to its immediate *sibling node* in the tree. The sibling node is the node immediately to the right of the current node. Because of the order of keys in the B + -Tree the sibling pointer always points to a node that has key values that are greater than the key values in the current node.

Order of a B + -Tree

- The *order* of a B + -Tree is the number of keys and pointers that an internal node can contain. An order size of *m* means that an internal node can contain *m-1* keys and *m* pointers.
- The order size is important because it determines how large a B + -Tree will become.
- For example, if the order size is small then fewer keys and pointers can be placed in one node and so more nodes will be required to store the index. If the order size is large then more keys and pointers can be placed in a node and so fewer nodes are required to store the index.

Searching a B+-Tree

Searching a B+-Tree for a key value always starts at the root node and descends down the tree. A search for a single key value in a B+-Tree consisting of unique values will always follow one path from the root node to a leaf node.



Searching for Key Value 6

· Read block *B3* from disc. *~ read the root node*

· Is *B3* a leaf node? No *~ its not a leaf node so the search continues*

· Is 6 <= 5? No *~ step through each value in B3*

· Read block B2. *~ when all else fails follow the infinity pointer*

· Is *B2* a leaf node? No *~ B2 is not a leaf node, continue the search*

· Is 6 <= 7? Yes ~ *6 is less than or equal to 7, follow pointer*

· Read block L2. ~ *read node L2 which is pointed to by 7 in B2*

· Is *L2* a leaf node? Yes ~ *L2 is a leaf node*

· Search *L2* for the key value 6. ~ *if 6 is in the index it must be in L2*

Searching for Key Value 5

· Read block *B3* from disc. ~ *read the root node*

· Is *B3* a leaf node? No ~ *its not a leaf node so the search continues*

· Is 5 <= 5? Yes ~ *step through each value in B3*

· Read block *B1*. ~ *read node B1 which is pointed to by 5 in B3*

· Is *B1* a leaf node? No ~ *B1 is not a leaf node, continue the search*

· Is 5 <= 3? No ~ *step through each value in B1*

· Read block *L3*. ~ *when all else fails follow the infinity pointer*

· Is *L3* a leaf node? Yes ~ *L3 is a leaf node*

· Search *L3* for the key value 5. ~ *if 5 is in the index it must be in L3*

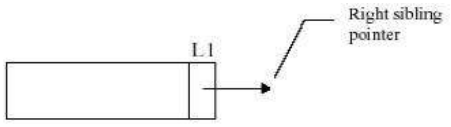
Inserting in a B+-Tree

A B+-Tree consists of two types of node: (i) leaf nodes, which contain pointers to data records, and (ii)internal nodes, which contain pointers to other internal nodes or leaf nodes. In this example, we assume that the order size is 3 and that there are a maximum of two keys in each leaf node.

Insert sequence: 5, 8, 1, 7, 3, 12, 9, 6

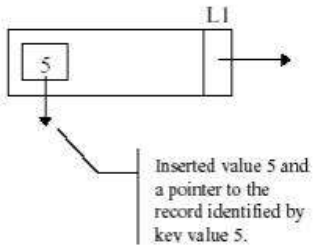
Empty Tree

The B+-Tree starts as a single leaf node. A leaf node consists of one or more data pointers and a pointer to its right sibling. This leaf node is empty.



Inserting Key Value 5

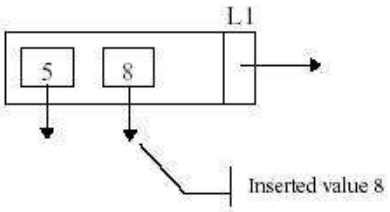
To insert a key search for the location where the key would be expected to occur. In our example the B+-Tree consists of a single leaf node, *L1*, which is empty. Hence, the key value 5 must be placed in leaf node *L1*.



Inserting Key Value 8

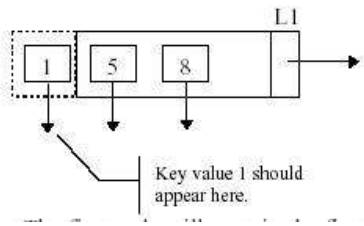
Again, search for the location where key value 8 is expected to be found. This is in leaf node *L1*.

There is room in *L1* so insert the new key.

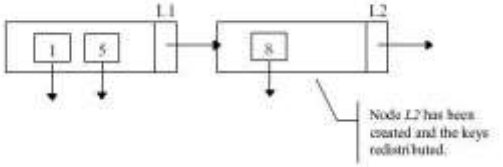


Inserting Key Value 1

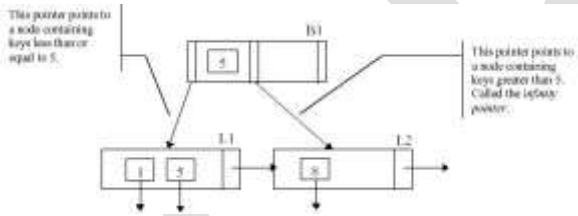
Searching for where the key value 1 should appear also results in *L1* but *L1* is now full it contains the maximum two records.



L1 must be split into two nodes. The first node will contain the first half of the keys and the second node will contain the second half of the keys



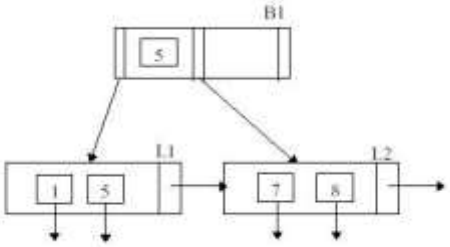
However, we now require a new *root* node to point to each of these nodes. We create a new root node and promote the rightmost key from node *L1*.



Each node is half full.

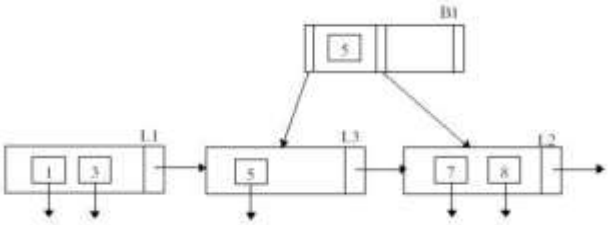
Insert Key Value 7

Search for the location where key 7 is expected to be located, that is, *L2*. Insert key 7 into *L2*.

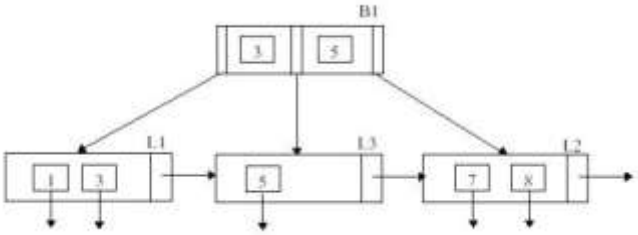


Insert Key Value 3

Search for the location where key 3 is expected to be found results in reading *L1*. But, *L1* is full and must be split.



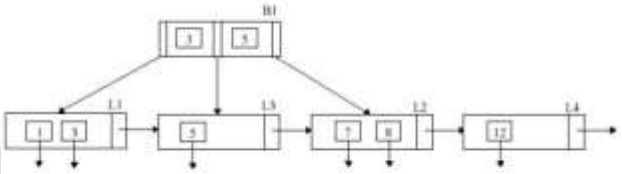
The rightmost key in *L1*, i.e. 3, must now be promoted up the tree.



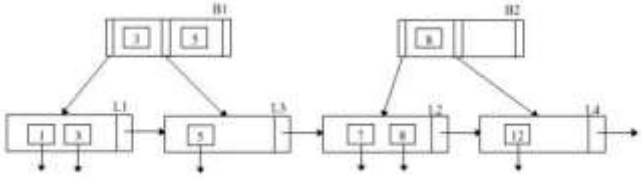
L1 was pointed to by key 5 in *B1*. Therefore, all the key values in *B1* to the right of and including key 5 are moved to the right one place.

Insert Key Value 12

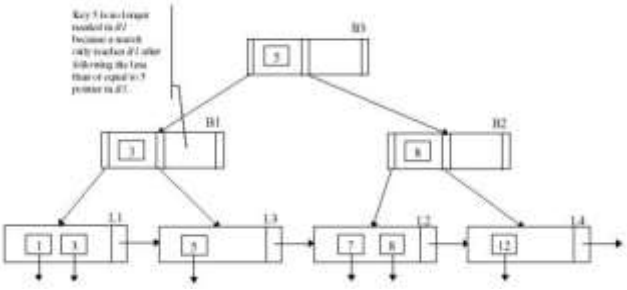
Search for the location where key 12 is expected to be found, *L2*. Try to insert 12 into *L2*. Because *L2* is full it must be split.



As before, we must promote the rightmost value of *L2* but *B1* is full and so it must be split.



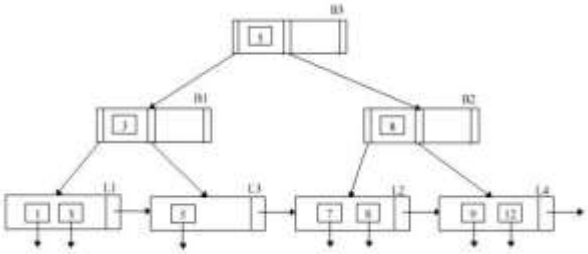
Now the tree requires a new root node, so we promote the rightmost value of *B1* into a new node.



The tree is still balanced, that is, all paths from the root node, *B3*, to a leaf node are of equal length.

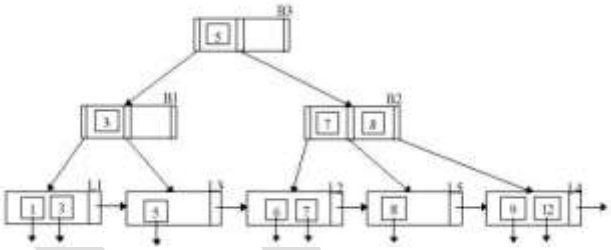
Insert Key Value 9

Search for the location where key value 9 would be expected to be found, *L4*. Insert key 9 into *L4*.



Insert Key Value 6

Key value 6 should be inserted into *L2* but it is full. Therefore, split it and promote the appropriate key value.



Leaf block *L2* has split and the middle key, 7, has been promoted into *B2*.

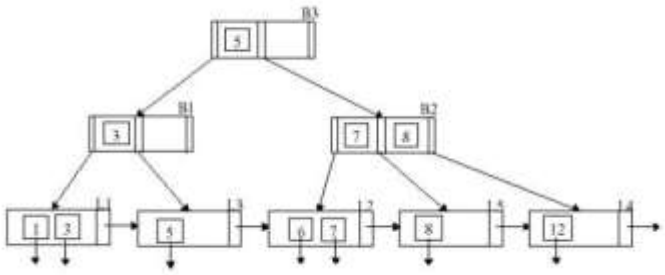
Deleting from a B+-Tree

Deleting entries from a B+-Tree may require some redistribution of the key values to guarantee a well-balanced tree.

Deletion sequence: 9, 8, 12.

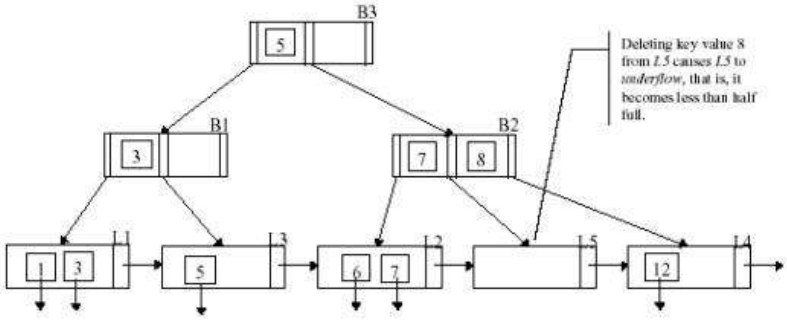
Delete Key Value 9

First, search for the location of key value 9, *L4*. Delete 9 from *L4*. *L4* is not less than half full and the tree is correct.

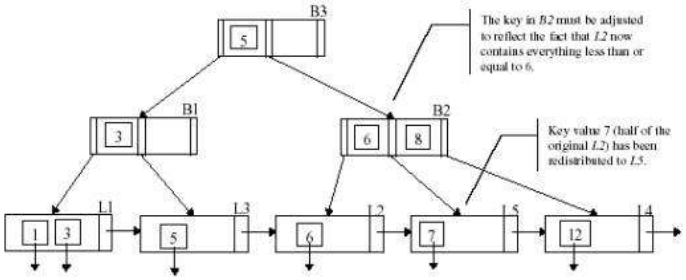


Delete Key Value 8

Search for key value 8, *L5*. Deleting 8 from *L5* causes *L5* to *underflow*, that is, it becomes less than half full.



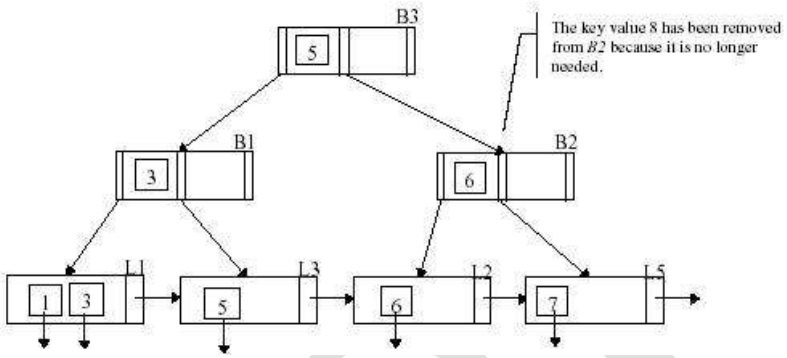
We could remove *L5* but instead we will attempt to redistribute some of the values from *L2*. This is possible because *L2* is full and half its contents can be placed in *L5*. As some entries have been removed from *L2*, its parent *B2* must be adjusted to reflect the change.



We can do this by removing it from the index and then adjusting the parent node *B2*.

Deleting Key Value 12

Deleting key value 12 from *L4* causes *L4* to underflow. However, because *L5* is already half full we cannot redistribute keys between the nodes. *L4* must be deleted from the index and *B2* adjusted to reflect the change.



The tree is still balanced and all nodes are at least half full. However, to guarantee this property it is sometimes necessary to perform a more extensive redistribution of the data.

Extendible Hashing

- In extendible hashing, a type of directory—an array of 2^d bucket addresses—is maintained, where *d* is called the **global depth** of the directory.
- The integer value corresponding to the first (high-order) *d* bits of a hash value is used as an index to the array to determine a directory entry, and the address in that entry determines the bucket in which the corresponding records are stored.
- However, there does not have to be a distinct bucket for each of the 2^d directory locations.
- Several directory locations with the same first *d* bits for their hash values may contain the same bucket address if all the records that hash to these locations fit in a single bucket.
- A **local depth *d***—stored with each bucket—specifies the number of bits on which the bucket contents are based.
- The value of *d* can be increased or decreased by one at a time, thus doubling or halving the number of entries in the directory array.
- Doubling is needed if a bucket, whose local depth *d'* is equal to the global depth *d*, overflows.
- Halving occurs if *d* > *d'* for all the buckets after some deletions occur.

- Most record retrievals require two block accesses—one to the directory and the other to the bucket.

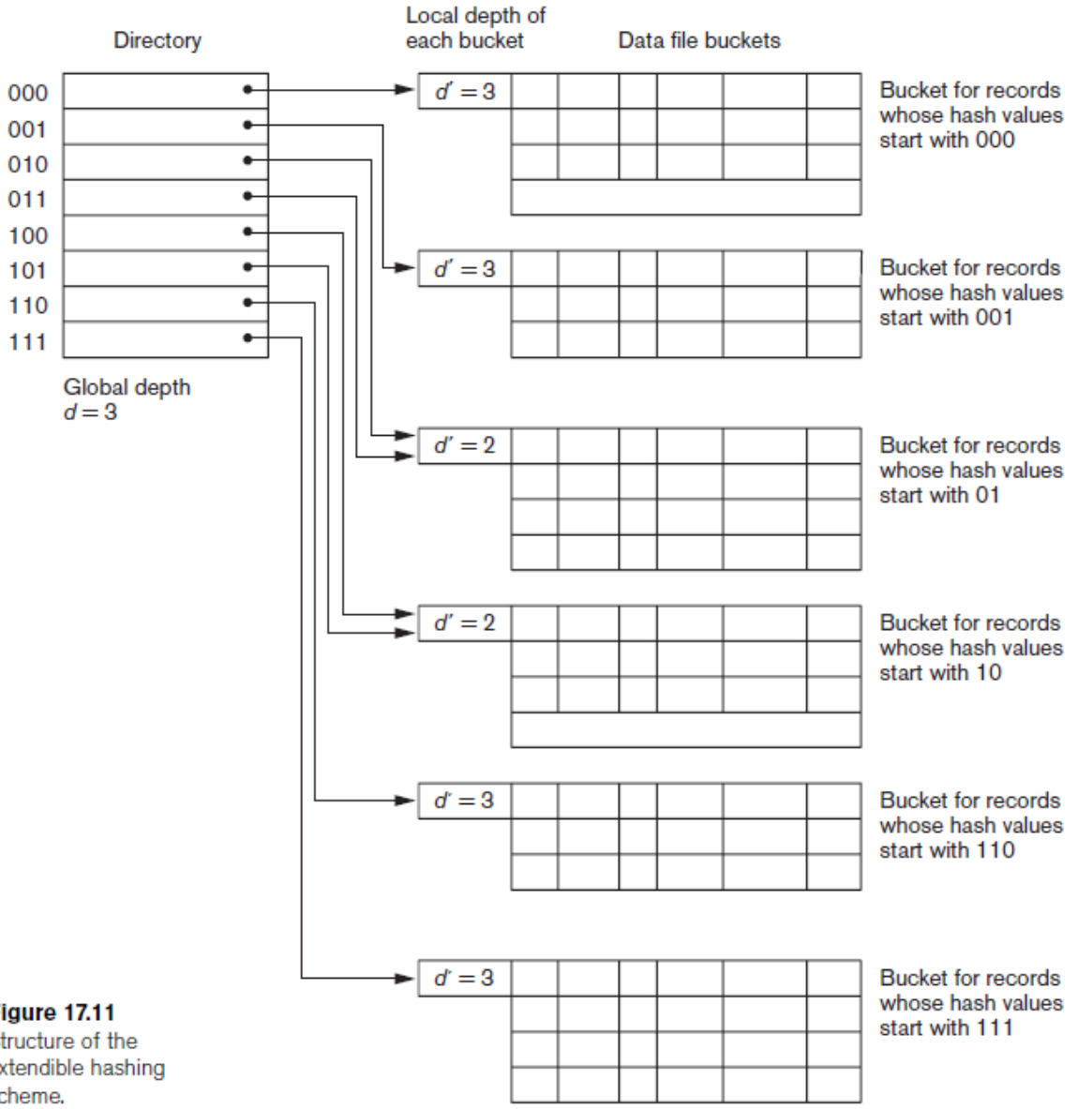


Figure 17.11
Structure of the extendible hashing scheme.

try it now

A KTU STUDENTS PLATFORM

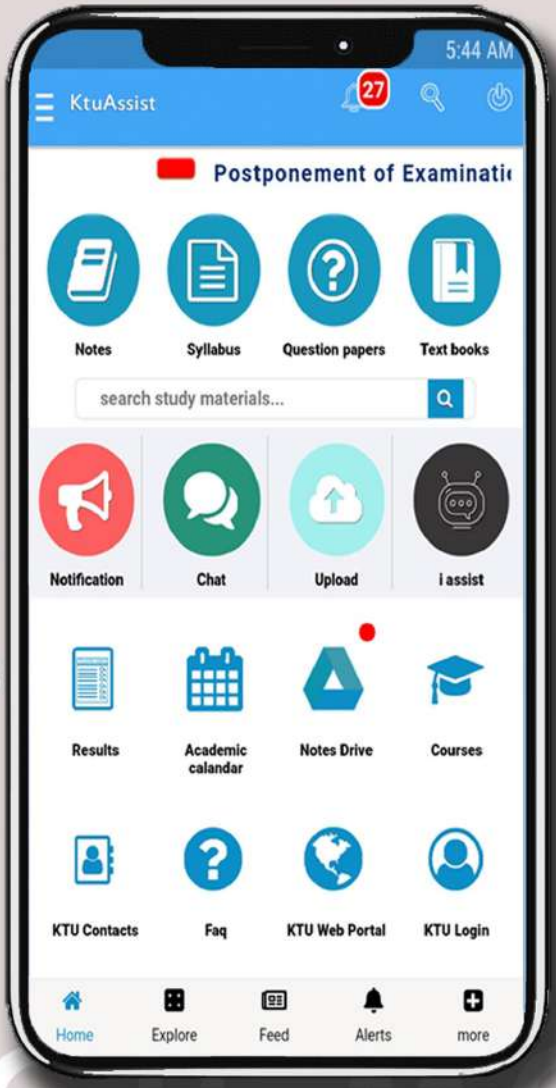
SYLLABUS

NOTES

TEXT BOOKS

QUESTION PAPERS

KTU NOTIFICATION



DOWNLOAD IT FROM GOOGLE PLAY

CHAT

FAQ

LOGIN

CALENDAR

MUCH MORE

DOWNLOAD APP

ktuassist.in

instagram.com/ktu_assist

facebook.com/ktuassist