Views

A view is a database object that is a logical representation of a table. It is delivered from a table but has no storage of its own and often may be used in the same manner as a table.

A view takes the output of the query and treats it as a table, therefore a view can be thought of as a stored query or a virtual table.

TYPES:

- Simple view
- Complex view
- Simple view can be created from one table where as complex view can be created from
- multiple tables.

Views

SQL> Create view dept_v as select deptno, sum(sal) t_sal from emp group by deptno;

SQL> Create view stud as select rownum no, name, marks from student;

SQL> Create view student as select *from student1 union select *from student2;

SQL> Create view stud as select distinct no,name from student;

Materialized Views

Use Case	Description
🔁 **Performance Improvement**	Speeds up complex joins, aggregations, and queries on large datasets.
⏱️ **Query Caching**	Stores the query result so it doesn’t need to be re-computed each time.
📊 **Data Warehousing**	Common in OLAP and reporting environments.
📦 **Replication**	Useful in distributed systems for replicating data across databases.

Materialized Views

```
CREATE MATERIALIZED VIEW mv_sales_summary
BUILD IMMEDIATE
REFRESH FAST ON COMMIT
AS
SELECT region, SUM(sales) AS total_sales
FROM sales
GROUP BY region;
```

INDEXES

Index is typically a listing of keywords accompanied by the location of information on a subject. We can create indexes explicitly to speed up SQL statement execution on a table. The index points directly to the location of the rows containing the value.

WHY INDEXES?

Indexes are most useful on larger tables, on columns that are likely to appear in where clauses as simple equality.

INDEXES

- **TYPES**
- Unique index
- Non-unique index
- Btree index
- Bitmap index
- Composite index
- Reverse key index
- Function-based index
- Descending index
- Domain index
- Object index
- Cluster index
- Text index

INDEXES

UNIQUE INDEX

Unique indexes guarantee that no two rows of a table have duplicate values in the columns that define the index. Unique index is automatically created when primary key or unique constraint is created.

Ex:

```
SQL> create unique index stud_ind on student(sno);
```

NON-UNIQUE INDEX

Non-Unique indexes do not impose the above restriction on the column values.

Ex:

```
SQL> create index stud_ind on student(sno);
```

INDEXES

BTREE INDEX or ASCENDING INDEX

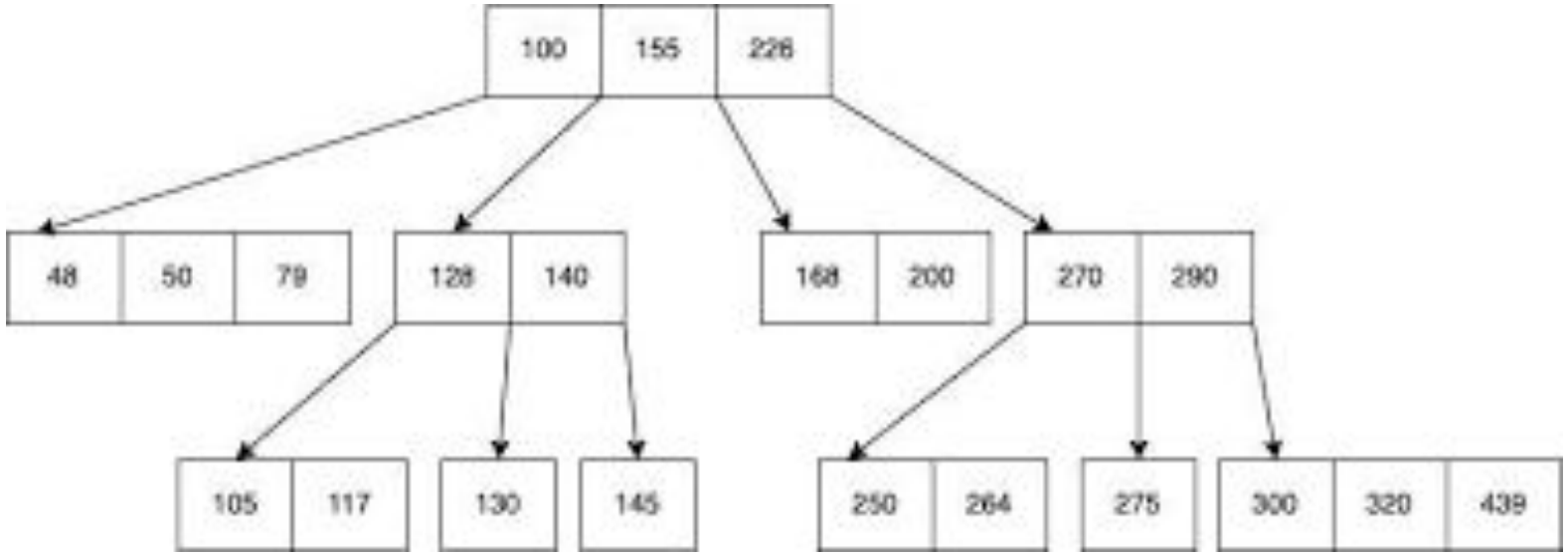
The default type of index used in an oracle database is the btree index. A btree index is designed to provide both rapid access to individual rows and quick access to groups of rows within a range. The btree index does this by performing a succession of value comparisons. Each comparison eliminates many of the rows.

Ex:

```
SQL> create index stud_ind on student(sno);
```

INDEXES

BTREE INDEX or ASCENDING INDEX



INDEXES

BITMAP INDEX

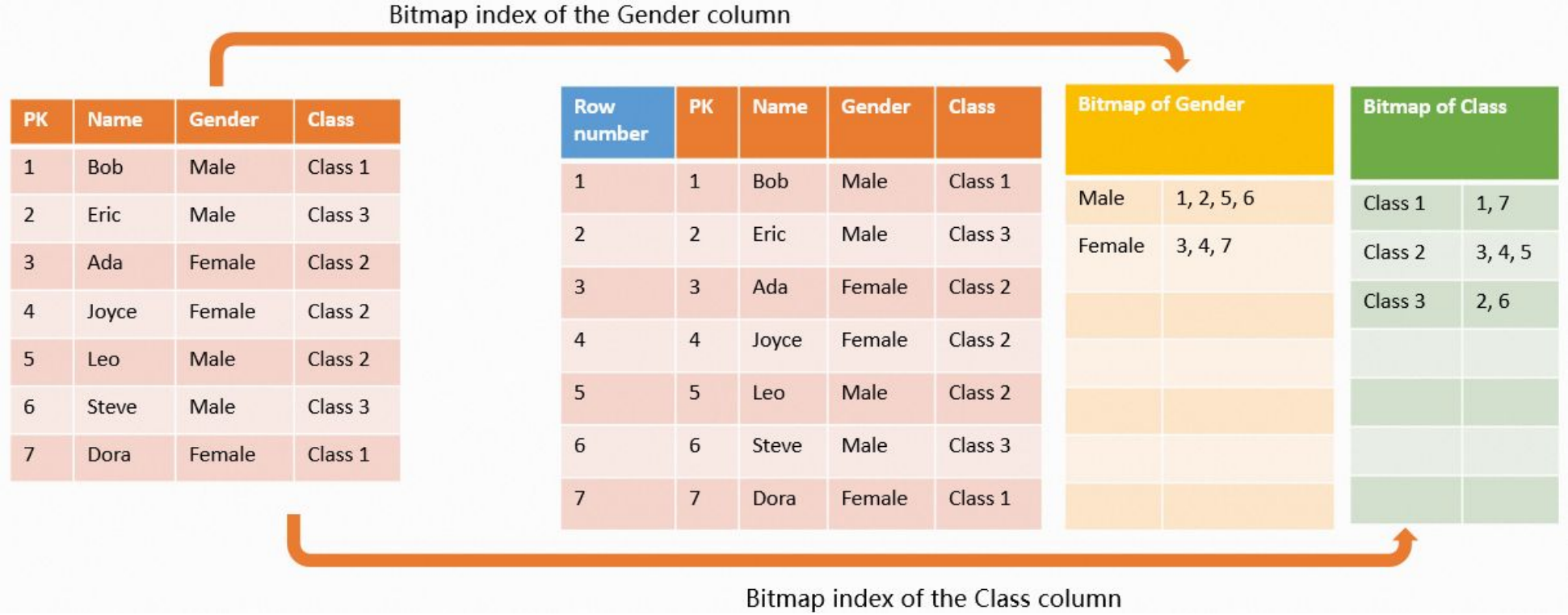
This can be used for low cardinality columns: that is columns in which the number of distinct values is snall when compared to the number of the rows in the table.

Ex:

```
SQL> create bitmap index stud_ind on student(sex);
```

INDEXES

BITMAP INDEX



PK	Name	Gender	Class
1	Bob	Male	Class 1
2	Eric	Male	Class 3
3	Ada	Female	Class 2
4	Joyce	Female	Class 2
5	Leo	Male	Class 2
6	Steve	Male	Class 3
7	Dora	Female	Class 1

Row number	PK	Name	Gender	Class
1	1	Bob	Male	Class 1
2	2	Eric	Male	Class 3
3	3	Ada	Female	Class 2
4	4	Joyce	Female	Class 2
5	5	Leo	Male	Class 2
6	6	Steve	Male	Class 3
7	7	Dora	Female	Class 1

Bitmap of Gender	
Male	1, 2, 5, 6
Female	3, 4, 7

Bitmap of Class	
Class 1	1, 7
Class 2	3, 4, 5
Class 3	2, 6

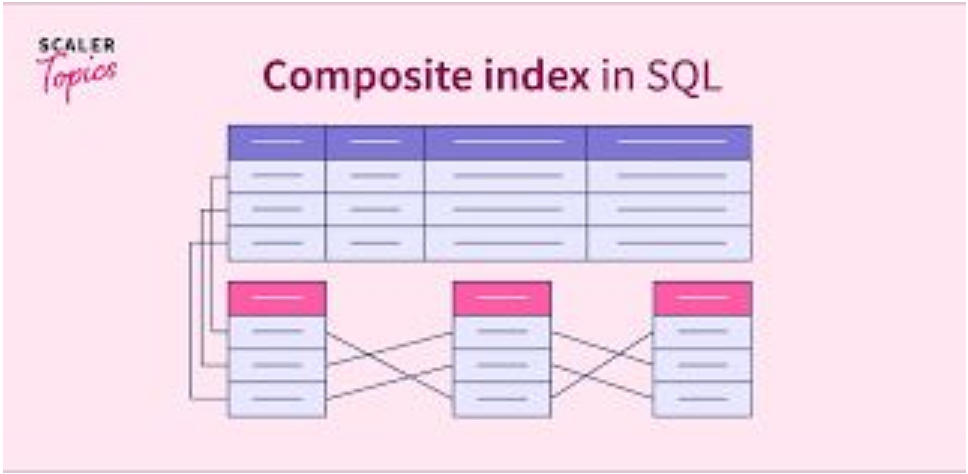
INDEXES

COMPOSITE INDEX

A composite index also called a concatenated index is an index created on multiple columns of a table. Columns in a composite index can appear in any order and need not be adjacent columns of the table.

Ex:

SQL> create bitmap index stud_ind on student(sno, sname);



INDEXES

REVERSE KEY INDEX

A reverse key index when compared to standard index, reverses each byte of the column being indexed while keeping the column order. When the column is indexed in reverse mode then the column values will be stored in an index in different blocks as the starting value differs. Such an arrangement can help avoid performance degradations in indexes where modifications to the index are concentrated on a small set of blocks.

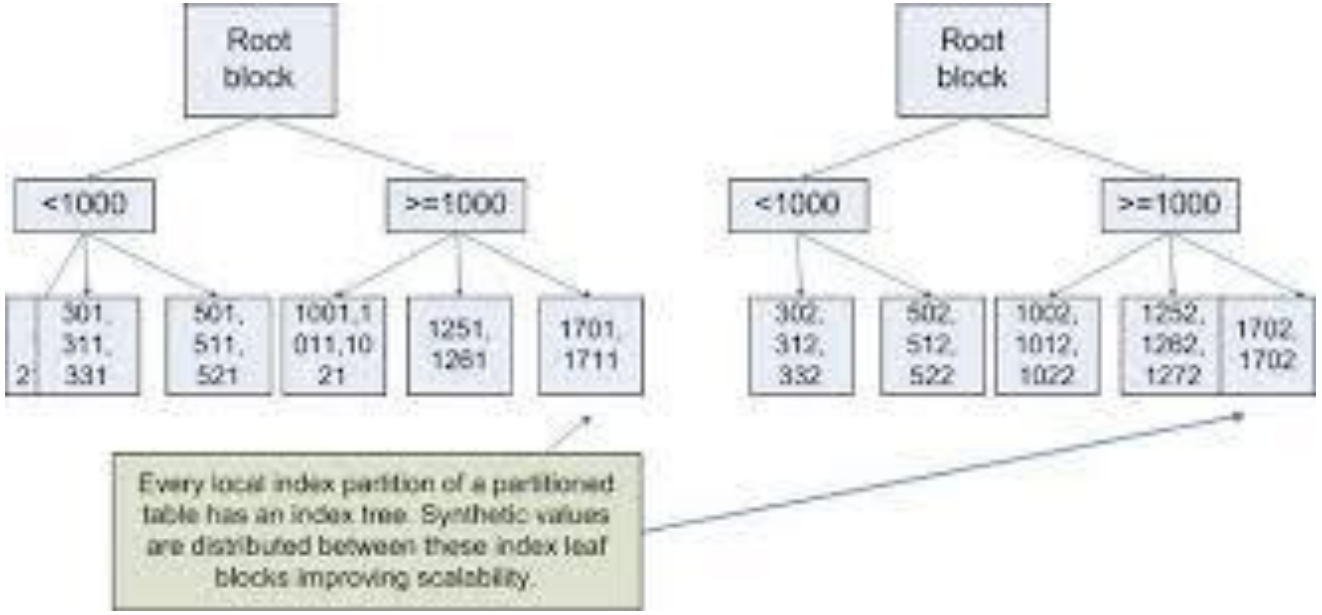
Ex:

```
SQL> create index stud_ind on student(sno, reverse);
```

We can rebuild a reverse key index into normal index using the noreverse keyword.

INDEXES

REVERSE KEY INDEX



INDEXES

FUNCTION BASED INDEX

This will use result of the function as key instead of using column as the value for the key.

view

Ex:

SQL> create index stud_ind on student(upper(sname));

FUNCTION BASED INDEX

This will use result of the function as key instead of using column as the value for the key.

view

Ex:

SQL> create index stud_ind on student(upper(sname));

INDEXES ----details

TEXT INDEX

Querying text is different from querying data because words have shades of meaning, relationships to other words, and opposites. You may want to search for words that are near each other, or words that are related to others. These queries would be extremely difficult if all you had available was the standard relational operators. By extending SQL to include text indexes, oracle text permits you to ask very complex questions about the text.

INDEXES ----details

HOW TO CREATE TEXT INDEX?

You can create a text index via a special version of the create index comman. For context index, specify the ctxsys.context index type and for ctxcat index, specify the ctxsys.ctxcat index type.

Ex:

Suppose you have a table called BOOKS with the following columns

Title, Author, Info.

SQL> create index book_index on books(info) indextype is ctxsys.context;

SQL> create index book_index on books(info) indextype is ctxsys.ctxcat;

INDEXES

MONITORING USE OF INDEXES

Once you turned on the monitoring the use of indexes, then we can check whether the table is hitting the index or not.

To monitor the use of index use the follwing syntax.

Syntax:

alter index index_name monitoring usage;

Schema Normalization in DBMS

Normalization is the process of organizing data in a database to eliminate **redundancy** and improve **data integrity**.

It is done by applying a series of **normal forms (NF)** — each form has rules that a table must follow.

Benefit	Description
✅ Removes redundancy	Avoids duplicate data
✅ Ensures consistency	Data is updated in one place
✅ Saves storage	Fewer repeating values
✅ Improves data integrity	Enforces meaningful relationships

Schema Normalization : First Normal Form (1NF)

First Normal Form (1NF)

A relation (table) is in **1NF** if:

Each **column contains values of a single type**.

There are no **repeating groups** or arrays.

Schema Normalization : First Normal Form (1NF)

Suppose you have a table like this:

StudentID	Name	Courses
1	Alice	Math, Science
2	Bob	English, History

Not in 1NF because Courses contains multiple values.
To convert to **1NF**, split the multi-valued attribute:

StudentID	Name	Course
1	Alice	Math
1	Alice	Science
2	Bob	English
2	Bob	History

Schema Normalization : Second Normal Form (2NF)

What is 2NF?

A table is in **2NF** if:

It is already in **1NF**.

No partial dependency exists (i.e., no non-prime attribute is dependent on a part of a candidate key).

Suppose we have:

StudentID	CourseID	StudentName	CourseName

Assuming StudentID + CourseID is the composite primary key.
Here, StudentName depends only on StudentID, not the full key — **partial dependency**.

Schema Normalization : Second Normal Form (2NF)

Fix: Decompose into two tables:

Students

StudentID	StudentName
1	Alice
2	Bob

Enrollments

StudentID	CourseID
1	101
1	102

Courses

CourseID	**CourseName**
101	**Math**
102	**Science**

Schema Normalization : Third Normal Form (3NF)

What is 3NF?

A table is in **3NF** if:

It is in **2NF**.

There are **no transitive dependencies** (i.e., non-prime attributes do not depend on other non-prime attributes).

Schema Normalization : Third Normal Form (3NF)

EmployeeID	Name	DeptID	DeptName

Here, DeptName depends on DeptID, not EmployeeID. So there's a **transitive dependency**.
Fix: Break into:

Employees

EmployeeID	Name	DeptID
1	John	10

Departments

DeptID	DeptName
10	HR

Schema Normalization : Boyce-Codd Normal Form (BCNF)

What is BCNF?

A stricter version of 3NF. A relation is in **BCNF** if:

It is in **3NF**.

Every determinant is a candidate key.

How to implement BCNF?

Professor	Subject	Department
Smith	DBMS	CS
Smith	OS	CS

Schema Normalization : Third Normal Form (3NF)

Assume:

- Each **professor teaches multiple subjects**.
- A **professor belongs to one department**.

So, Professor → Department (not a candidate key → violates BCNF).

Professors

Professor	Department
Smith	CS

Teaches

Professor	Subject
Smith	DBMS
Smith	OS

CONSTRAINTS

Constraints are categorized as follows.

Domain integrity constraints

- Not null
- Check

Entity integrity constraints

- Unique
- Primary key

Referential integrity constraints

- Foreign key

NOT NULL AND CHECK

Not Null

SQL> create table student(no number(2) not null, name varchar(10), marks number(3));

CHECK

This is used to insert the values based on specified condition.

We can add this co nstraint in all three levels.

SQL> create table student(no number(2) , name varchar(10), marks number(3) check

(marks > 300));

We can add constraints in three ways.

Constraints are always attached to a column not a table.

Column level -- along with the column definition

Table level -- after the table definition

Alter level -- using alter command

All three Levels

COLUMN LEVEL

SQL> create table student(no number(2) , name varchar(10), marks number(3) check

(marks > 300));

TABLE LEVEL

SQL> create table student(no number(2) , name varchar(10), marks number(3), check

(marks > 300));

ALTER LEVEL

SQL> alter table student add check(marks>300);

UNIQUE

This is used to avoid duplicates but it allow nulls.

We can add this constraint in all three levels.

COLUMN LEVEL

SQL> create table student(no number(2) unique, name varchar(10), marks number(3));

TABLE LEVEL

SQL> create table student(no number(2) , name varchar(10), marks number(3),

unique(no));

ALTER LEVEL

SQL> alter table student add unique(no);

PRIMARY KEY

This is used to avoid duplicates and nulls. This will work as combination of unique and not null.

Primary key always attached to the parent table.

We can add this constraint in all three levels.

PRIMARY KEY

COLUMN LEVEL

SQL> create table student(no number(2) primary key, name varchar(10), marks number(3));

marks number(3));

TABLE LEVEL

SQL> create table student(no number(2) , name varchar(10), marks number(3),

primary key(no));

ALTER LEVEL

SQL> alter table student add primary key(no);

FOREIGN KEY

This is used to reference the parent table primary key column which allows duplicates.

Foreign key always attached to the child table.

We can add this constraint in table and alter levels only.

TABLE LEVEL

SQL> create table emp(empno number(2), ename varchar(10), deptno number(2),

primary key(empno), foreign key(deptno) references dept(deptno));

TABLE LEVEL

SQL> create table emp(empno number(2), ename varchar(10), deptno number(2),

primary key(empno), foreign key(deptno) references dept(deptno));

USING ALTER

- **ADDING COLUMN** :alter table <table_name> add <col datatype>;
- **REMOVING COLUMN** :alter table <table_name> drop <col datatype>;
- **INCREASING OR DECREASING PRECISION OF A COLUMN:**

  alter table <table_name> modify <col datatype>;
- **DROPPING UNUSED COLUMNS**: alter table <table_name> drop unused columns;
- **RENAMING COLUMN** : alter table <table_name> rename column <old_col_name> to <new_col_name>;

Joins with Alias

```
select e.id,d.id from emp1 e,dept d where e.id=d.id;
```

EXPLAIN PLAN

EXPLAIN PLAN shows the **steps and order** Oracle will follow to execute a SQL statement — including things

```
EXPLAIN PLAN FOR
SELECT emp_id, name, salary, LEAD(salary) OVER (ORDER BY emp_id) AS next_salary FROM employees;

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```