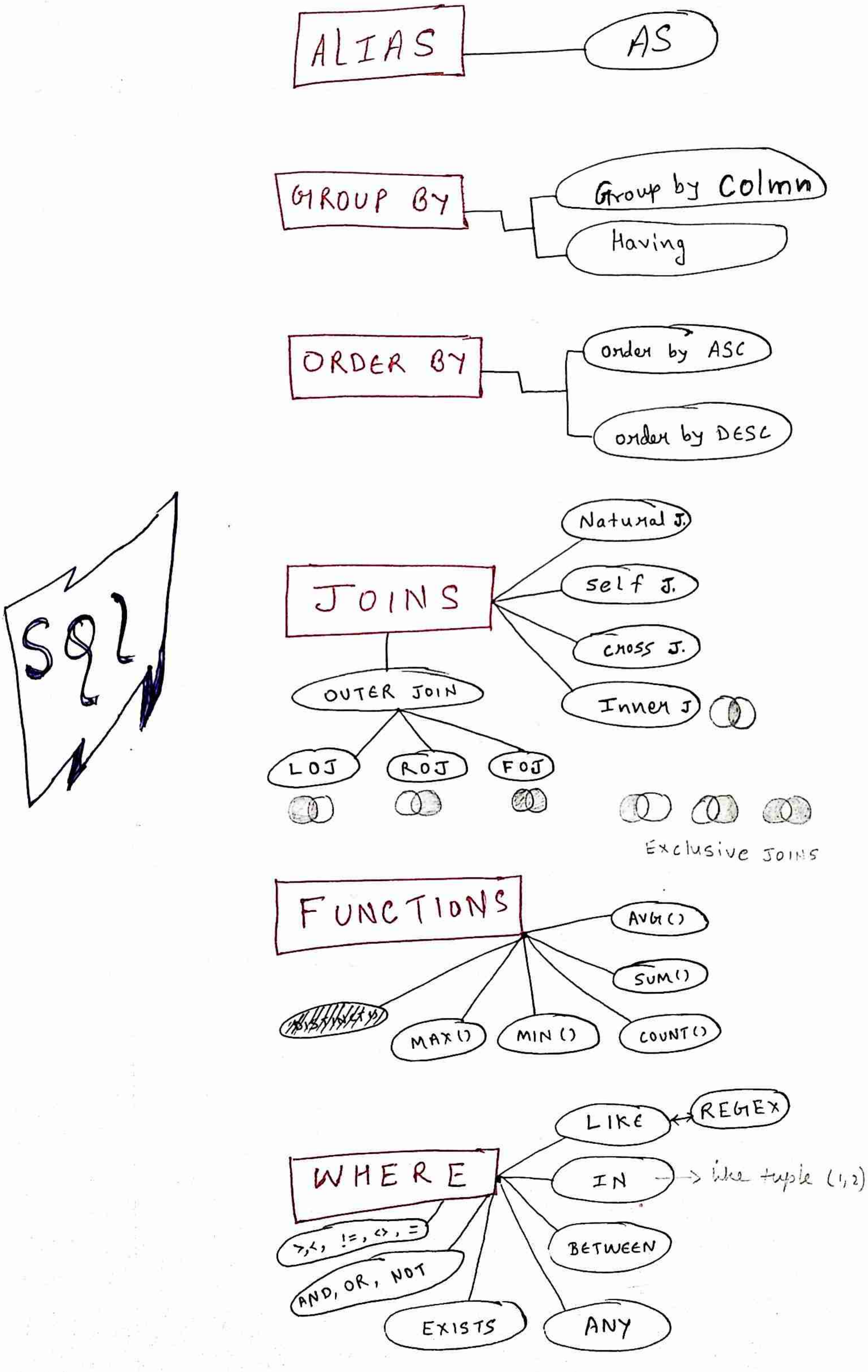
SQL
ALIAS
AS
GROUP BY
Group by Colmn
Having
ORDER BY
order by ASC
order by DESC
JOINS
Natural J.
self J.
CROSS J.
Inner J
OUTER JOIN
LOJ
ROJ
FOJ
Exclusive JOINS
FUNCTIONS
AVG()
SUM()
MAX()
MIN()
COUNT()
WHERE
LIKE
REGEX
IN
like tuple (1,2)
BETWEEN
ANY
EXISTS
AND, OR, NOT
>,<, !=, <>, =

SQL Commands

DDL

1 CREATE

2 ALTER – ADD / DROP / MODIFY

5 DROP

3 RENAME

4 TRUNCATE

defines & manages structure of a database

C
A
R
T
DROP

DML

SELECT

INSERT

UPDATE

DELETE undo able

~~MERGE~~

manipulation & retrieval of data stored in a database

DCL

GRANT

REVOKE

for controling access to the database

TCL

COMMIT

ROLLBACK

SAVEPOINT

Transaction

INDEXING

to speed up Query exec^n

Select
from
join
on
where
group by
having
order by

} Optimize Query Execut^{n}

Let's start...

ALIAS

```
SELECT   column_name AS "alias_name"
FROM     table_name ;
```

GROUP BY

i) Group by Column

```
SELECT     column_name(s)
FROM       table_name
WHERE      condtn
GROUP BY   column_name (s)
ORDER BY   column_name (s) ;
```

ii) HAVING (condition)

```
SELECT     ~~*~~ column_name (s)
FROM       table_name
WHERE      condition
GROUP BY   column_name(s)
HAVING     condition
ORDER BY   column_name(s)
```

eg-
```
SELECT COUNT (name) , city
FROM students
GROUP BY city
HAVING max(marks) > 90;
```

used when we want to apply any condition after grouping

ORDER BY

used to sort result set in ASC/Desc order

```
SELECT    column1, column2 ....
FROM      table_name
ORDER BY  column 1, column 2, ... ASC/DESC ;
```

JOINS

why → ? Normalization, breaking tables, result → info gets stored in multiple tables.

i) Natural Join

⇒ no condition is needed to join two or more tables, it automatically joins them based on common column.

```
SELECT  *
FROM    table1_name
NATURAL JOIN  table2_name ;
```

⇒ be careful while using NATURAL JOIN.

— ~~it compares all columns to find common~~

Drawbacks

— 1) If two tables have multiple common ~~col~~, columns, NATURAL JOIN FAILS.

— 2) If two tables have same column "date", it will Fail if the values of rows are different.

main drawback

Ambiguity :- when there are two or more columns with the same name in the two tables being joined.

In this case, the dB engine can't determine which column to use for the join. Hence error thrown.

student

Student_id	Name	Age
1	Alice	20
2	Boby	22
3	Carol	21

course

Course-id	C-Name	student_id
101	Math	1
102	History	2
103	Biology	1

Student-id	Name	Age	Course-id	C-Name
1	Alice	20	101	Math
2	Boby	22	102	History
1	Alice	20	103	Biology

Select *
FROM student
NATURAL JOIN course

ii) SELF JOIN

→ table is joined with itself.

→ when you need to compare rows within same table

→ Eg - Find all employees who have the same manager.

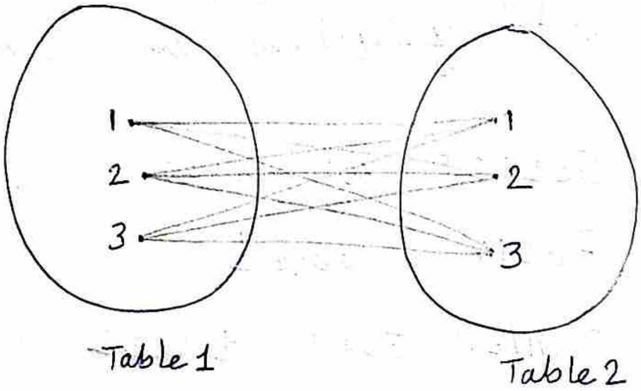
- find all customers having same shiping address

```
SELECT
        e.employee-id,
        e.first_name,
        m.first_name As "manager"
FROM employees e
SELF JOIN employees m
    ON e.reports_to = m.employee-id
```

employee_id	first_name	manager

iii) CROSS JOIN (Cartesian Join)

→ used to combine each row of one table with each row of another table.



→ when to use CROSS JOIN ?

⇒ You have two columns : size & color and You need ~~to~~ a result set to display all possible paired combinations of those.

```
SELECT      column(s)
FROM        table1
CROSS JOIN  table2;
```

Car_model

1	Camry
2	Corola
3	Prius

CROSS JOIN

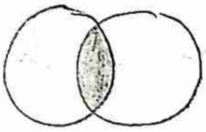
color

1	Black
2	Red
3	silver

	car_model	color
1	camry	Black
2	camry	Red
3	camry	silver
4	corola	Black
5	corola	Red
6	corola	silver
7	⋮	⋮
8		

Default Join

iv) INNER JOIN



⇒ returns records (rows) that have matching values in both tables.

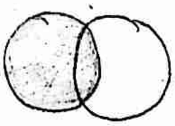
```
SELECT  column(s)
FROM    table 1
INNER JOIN  table 2
ON  table1. col_name  =  table 2. col_name;
```

v) Outer Join

Outer Join → Left outer Join
→ Right outer join
→ full outer join

a) LEFT OUTER JOIN

optional

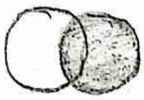


gives all records from left table as well as the matched & rows from right table.

```
SELECT  column(s)
FROM    table 1
LEFT ~~JOIN~~  OUTER  JOIN  table 2
ON  table1. col_name  =  table2 . col_name;
```

ⓑ RIGHT OUTER JOIN

optional

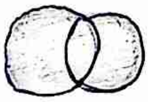


gives all records (rows) of right table and the matching rows from left table.

```
SELECT  column(s)
FROM
        table 1
RIGHT OUTER JOIN  table 2
ON table1. col_name  = table 2 · col_name;
```

ⓒ FULL OUTER JOIN

optional



LEFT JOIN
UNION
RIGHT JOIN

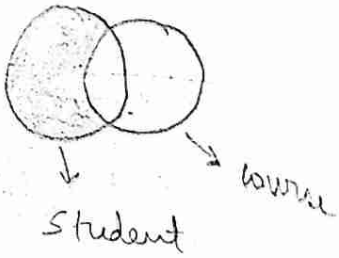
returns all records when there is a match in either left or right table.

```
SELECT  *  FROM  student as a
LEFT JOIN  course  as b
ON  a·id  =  b·id
UNION
SELECT  *  FROM  student as a
RIGHT JOIN  course  as b
ON  a·id  =  b·id ;
```

UNION → gives only unique values

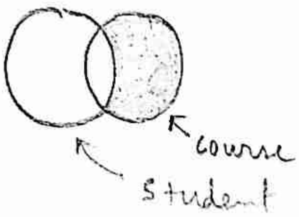
Extra JOINS

① Left Exclusive Join



```
SELECT  *
FROM student as a
LEFT JOIN course as b
ON a.id = b.id
WHERE b.id IS NULL ;
```

② Right Exclusive Join



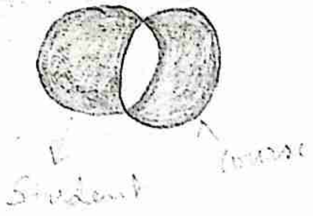
```
SELECT  *
FROM   course as a
RIGHT JOIN student as b
ON  a.id = b.id
WHERE a.id  IS NULL ;
```

③ FULL Exclusive JOIN

LEFT Exclusive JOIN

UNION

RIGHT EXCLUSIVE JOIN



Keys

row, t, r
col, a, field/friend

~~constraints : used to give rules for data in a table.~~

why :- to get records ASAP.

1) Primary key :

- A column that uniquely identifies a record (row) in the table.
- has unique data
- can't have NULL values.
- only one primary key is allowed in a table

2) Foreign key :

- A column that is a primary key of another table.
- used to ~~mat~~ link two or more tables.
- a table can have multiple FK.

3) Composite key :-

```
PRIMARY KEY (colA, colB);
```

A primary key that is made by the combination of more than one column is called Composite key.

when a single column is not enough to uniquely identify a row, then two (or more) columns are made primary key known as composite key.

4) UNIQUE KEY

- just like primary key except, it can accept only one NULL value.

- ~~A~~ multiple unique key can be created in a table.

Eg -

```
CREATE TABLE student(
        s-id int NOT NULL,
        Name varchar(25) NOT NULL,
)       UNIQUE (s-id);
```

5) Alternate Key

A column which is not a primary key is called alternate key.

or

All columns except primary key column are alternate keys.

Functions

1) AVG()

eg-
```
SELECT  AVG(col-name)
FROM    table-name
WHERE   condition;
```

2) SUM()

3) COUNT()

(4) MIN()

(5) MAX()

WHERE

(1) LIKE (old method)

```
SELECT *
FROM   table-name
WHERE  column LIKE 'pattern'
```

9 → '%9'

__3...... → '__3%'

...A.... → '%A%'

* REGEXP clause. (replcmt for LIKE)

```
SELECT *
FROM   customers
WHERE  phone LIKE '%9';
```

```
SELECT *
FROM   customers
WHERE  phone
  REGEXP 'g';
```

Symbols.

^	→ starting string	⇒	'^Mr'
$	→ ending string	⇒	'Kumar$'
|	→ OR (pipe)		
[ ]	→ must include any or all characters.		

Mr Ram Kumar

Q. Use REGEXP, to get customers whose -

i) first names are ELKA or AMBUR
ii) last name ends with EY or ON
iii) last name starts with My or contains SE
iv) last name contains B followed by R or U

Solⁿ

```
i) WHERE first_name REGEXP '^ELKA|^AMBUR'
ii) WHERE last_name REGEXP 'EY$|ON$'
iii) WHERE last_name REGEXP '^My|SE'
iv) WHERE last_name REGEXP 'B[RU]'
```

Eg - [gim]e

→ any one including e
or
→ any two including e
or
→ any three including e

* LIMIT

```
SELECT *
FROM customers
LIMIT 5, 3;
```

5th row तक skip करके 6,7,8th row दिखाएगा

Top 3 most loyal customers :-

```
SELECT *
FROM customers
ORDER By points DESC
LIMIT 3;
```

WHERE

2) IN

```
SELECT *
FROM customers
WHERE state IN('VA','FL','GA')
```

3) BETWEEN

```
SELECT *
FROM customers
WHERE birth-date BETWEEN '1990-01-01' AND
                         '2000-01-01';
```

4) ANY

```
SELECT ProductName
FROM products
WHERE product_id = Any (
                        SELECT product-id
                        FROM order_details
                        WHERE Quantity = 10
                   );
```

5) > , >= , < , <= , = , != <>

same

```
SELECT *
FROM customers
WHERE points > 3000
```

SQL Commands

1. Database

```
CREATE DATABASE db_name;
DROP DATABASE db_name;
```

```
CREATE DATABASE IF NOT EXISTS db-name;
DROP DATABASE IF EXISTS db_name;
```

```
USE db_name;
SHOW DATABASES;
```

```
RENAME DATABASE old_name
  TO new_name;
```

TABLE related

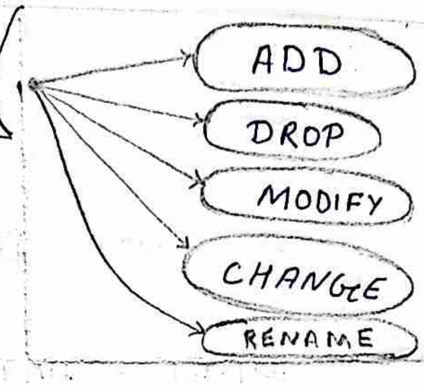
1) DDL (CART Drop)

i) CREATE TABLE

```
CREATE TABLE ~~emp~~ table_name (
    col1  datatype  constraint,
    col2  datatype  constraint,
       :
);
```

DDL

ii) ALTER TABLE



Alter table tableName
Rename column columnName TO new-columnName

ADD (column)

ALTER TABLE table_name
ADD column_name datatype constraint;

DROP (column)

ALTER TABLE table_name
DROP COLUMN ~~column_name~~ column_name;

~~MODIFY (structure)~~

ALTER TABLE table_name
[ADD column_name datatype]
[DROP column_name]
[MODIFY column_name datatype]
[~~RENAME column_name TO new_col_name~~]
[CHANGE old-col_name new-col_name datatype]

→ can't rename TABLE NAME using ALTER

→ can't change datatype of columns which have data.

→ can't drop a column that has a foreign key ~~co~~ reference in other table.

DDL

iii) RENAME TABLE/VIEW/DATABASE

```
RENAME old_name TO new_name;
```

Eg- RENAME TABLE customers
TO clients;

~~Table can be renamed using RENAME TABLE as well as ALTER TABLE~~

~~ALTER TABLE table_name~~
~~RENAME old_name TO new_name;~~

iv) TRUNCATE TABLE

deletes all data from table but not the table.

```
TRUNCATE TABLE table_name;
```

v) DROP TABLE

→ deletes a table in the dB.
→ can't be undone.

```
DROP TABLE employees;
```

* DELETE FROM (DML का है)

→ deletes data/records/rows from a table
→ can be undone (recovered)

DML

can be rolled back

2) DML

SELECT
INSERT
DELETE
UPDATE } [illegible]

i) SELECT

```
SELECT column_name(s)
FROM   table_name;
```

ii) INSERT (rows)

```
INSERT INTO tableName (col1, col2, ...)
VALUES (value1, value2, ...);
```

iii) DELETE FROM (deletes rows)

```
DELETE FROM table_name
WHERE condition;
```

```
DELETE *
FROM table1;
```

all rows gone

iv) UPDATE (existing rows)

```
UPDATE table_name
SET    col1 = val1, col2 = val2, ...
WHERE  condition;
```

Eg -
```
UPDATE customer
SET customerName = "Ram"
WHERE age = 22;
```

DCL

3) DCL

-> used to control access to data in a dB.

i) GRANT

-> grant SELECT permission to the user1 on the customers table

```
GRANT SELECT
 ON   customers
 TO   user1 ;
```

ii) REVOKE

-> revoke/ remove UPDATE permission from user2 on the orders table.

```
REVOKE  UPDATE
 ON     orders
 FROM   user2 ;
```

TCL

4) TCL

i) COMMIT →

```
COMMIT ;
```

ii) ROLLBACK →

```
ROLLBACK ;
```

iii) SAVEPOINT

→ can be used to -

⇒ roll back changes

⇒ recover from errors that occured during a transaction

~ create SAVEPOINT

```
SAVEPOINT my-savepoint ;
```

~ Rollback to savepoint "my-savepoint"

```
ROLLBACK TO SAVEPOINT my-savepoint
```

VIEW

- → A virtual table that is based on the result-set of an SQL statement.
- → not stored in dB as tables but are stored in data dictionary.
- → used to hide sensitive data
- → A view always shows up-to-date data. The dB engine recreates the view, every time a user queries it.

---

syntax:

```
CREATE VIEW view_name AS
SELECT column(s)
FROM table_name
WHERE condition;
```

Eg:-

```
             CREATE VIEW cust_with_orders AS
             SELECT customers.name, orders.order_id
             FROM customers
inner join → JOIN orders
               ON customers.id = orders.customer_id;
```

Sub-Queries

→ A subquery / nested query / inner query is a query inside another SQL query.

→ It involves two SELECT stmts.

→ outer query → Parent query, inner query → child query

→ used to filter data.

→
```
SELECT   column(s)
FROM     table_name
WHERE
         colname   operator
 ( subquery);
```

Eg - point all customers who have placed orders.

```
SELECT   customers.name
FROM     customers
WHERE
         customers.id  IN(
    SELECT
              orders.customer_id
);  FROM  orders
```

Q. Select the name of customer who has placed the most orders.

```
SELECT   customers.name
FROM     customers
WHERE    customers.id = (
     SELECT   orders.customer_id
     FROM     orders
     GROUP BY  orders.customer_id
     ~~ORDER BY  orders.customer_id~~
     ORDER BY  COUNT (*) DESC
     LIMIT 1
);
```

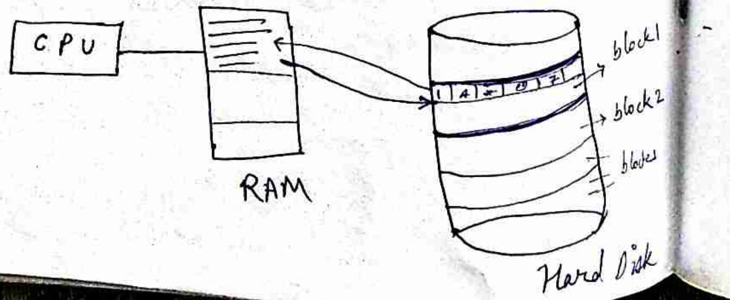
Indexing

Q: Why Indexing is used ?
→ to speed up the query exect^{n}.

If we have a book with 500 pages, and has no index page in it, and we are asked to find a particular heading, we would end up searching the entire 500 pages or may be it is available at the first page.

Average ~~time~~ pages would be 250 pages to get ~~search~~ that heading.

If we had an index ~~Ex~~, it would be searched (found) within 2-3 pages only.



Similarly in Computers.

when a sql query is fired, ~~the~~ CPU asks the RAM to give it the tables that are stored in Hard disk and processes them to get d reqd data.

The hard disk is logically divided into blocks.

At a time, ~~the~~ RAM accesses a particular block and serves it to the CPU, if the data is found then its hit otherwise miss.

This process continues until d. reqd data is not found.

This process ~~takes~~ of accessing each block of HD takes significant amount of time.

~~If~~ This time can be reduced by using indexing concept.

When there is index, the RAM will access that block only which has the reqd data, after looking into the index.

Hence query execution becomes faster.

Linear search (unordered | unarranged)
takes O(n) time to search .

Binary search (ordered | sorted data only)
takes ~~log~~ $O(\log_2(N))$ time to search.

Indexing बोलता है $\log_2 N$ से भी कम time में search करके ढूंढेगा ।

---

* Index table is also stored in the HD but RAM loads the Index table first.

Index table contains (Key - pointer) pair columns (like dictionary in Python)

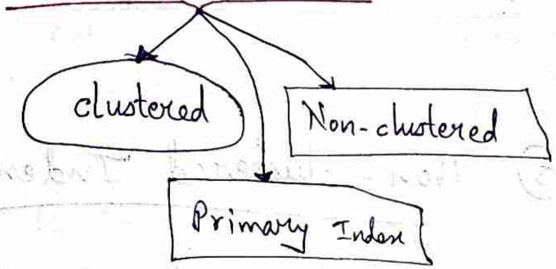
eg.

Key (RollN)	Pointer
2201110	#C305.

→ locations of block inside HD

⇒ Only those columns can be indexed which has UNIQUE ~~KEY~~ (PRIMARY KEY) values

Types of Index



① PRIMARY INDEX (clustered index ही है)

A unique index that is automatically created when a primary key is defined.

② CLUSTERED INDEX

- only one index in a table i.e PKey
- The rows in the table are physically sorted based on the order of the index key
- benefit for queries that involve sequential access to data.
- ~~but~~ slow for queries like insert, update & delete because the dB need to physically rearrange the data.

~~③ Nonclustered Index~~

③ Non-clustered Index

- A table can have multiple non-clustered index
- It is a separate ~~data~~ structure that contains a copy of the indexed columns & a pointer to the actual data rows. (logical sorting)
- The columns included in a non-clustered index can be different from the primary key.
- improves performance of queries that involve filtering, sorting or joining data.

⇒

```
CREATE  CLUSTERED  INDEX  cid
ON   Employee ( EmployeeID ) ;
```

cid ⇒ clustered index name
Employee ⇒ name of table
EmployeeID ⇒ column on which clustered index is created.

→ If the table already has a clustered index, then it **FAILS**

⇒

```
CREATE  NONCLUSTERED  INDEX  ncid
ON   Orders ( cust_id , order_date ) ;
```

ncid ⇒ nonclustered index name
Orders ⇒ name of table
cust_id , order_date ⇒ columns on which the NC index is made.

① Disadvantages of Indexing

→ increased storage space

→ managing and maintaining a large number of indexes can become challenging.