① **Distinct** keyword operates on a column.

```
select distinct(column)
    from table
```

② **count**
- returns i/p rows matching a condition.

```
select count(*) from table;
```

③ **where**
- condition on columns for returning rows.

```
select c1, c2
    from table
    where condition
```

3.1 Comparison operators
=, >=, <=, !=, >, <

3.2 Logical operators
- AND, OR, NOT
- combine multiple comparison operators

④ **order by**

```
select column from table
    order by column

select c1, c2 from table
    order by, c1, c2

select c2, c2 from table
    order by c1 ASC,
        c2 DESC
```

⑤ **Limit**

```
select c2 from table
    order by c2 DESC
    Limit 10;
```

⑥ **Between** (operator)
- match value against range of values.

Between LOW and HIGH
- (NOT)

* ISO 8601 — std date format
  - YYYY-MM-DD
  - YYYY-MM-DD

⑦ **In**
- condition that checks for a value in list of multiple options

value IN (op1, op2, op3, op4)

⑧ **LIKE** and **ILIKE**
- general pattern in a string

pattern matching

* wildcard character %

where name like 'A%'
(Names that begin with A)

Note: Like is case sensitive

⑨ _ Underscroll
- Allow to replace single char

where value LIKE 'version#__'

where name LIKE '_her%'

% → Any number of characters (open)

_ → only sing character

* ILIKE – makes it case-Insensitive

(10) Group by → categorical column

↓

– Aggregate data and
– Apply functions

Take multiple inputs and return a single output

Avg() , sum()
max()
min()
count()
min()

can be applied after

select clause (OR) Having clause

(11) Group by

↓

– Appear after from/where state.
– can 'group by' multiple coln.

```
select c1, c2, sum(c3)
from table
where c1 != A,
Group by c1, c2
```

Note: 'where' statements should not refer to aggregate ~~statem~~ result

(12) DATE

↓

returns date from column and deletes everything else

SELECT DATE (column_date)
from table

(13) Having

↓

- Filters after an aggregat^n has taken place.
- X we cannot use where, to filter based off of aggregate results
  ∵ Those happen after where is executed
- use aggregate result as after along with GROUP BY

(1) Avg

Round off

```
select Round (avg (c2), 2)
from film;
```

decimal

```
*select category_col,
        AGG (data_col),
from table
Group by category_col
```

(14) AS

↓

Allows to create alias for column / result

→ SELECT column AS new_name
FROM table

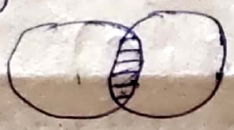
* Can't use AS inside WHERE operator.

* AS gets executed at a very end of query. — to get filtered

(15) Inner Join

↓

- combine multiple tables together



match in both tables

→ SELECT * FROM tableA
INNER JOIN tableB
ON tableA.col_match
= tableB.col_match

(16) Full Outer Join

↓

specify how to deal with values present only in one of tables being joined

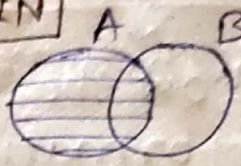
* Full Outer Join with 'WHERE'

↓

Get rows - unique to either table

→ SELECT * FROM table A
FULL OUTER JOIN table B
ON tableA.colm = tableB.col_m
Where tableA.col is NULL
OR table B.col IS NULL

(17) LEFT OUTER JOIN



→ SELECT *
FROM tableA
LEFT OUTER JOIN tableB
ON tableA.col_m = tableB.col_m

* WHERE

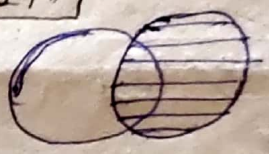
↓

Unique Entries to Left table

WHERE tableB.col IS null

(18) RIGHT JOIN

RIGHT JOIN



WHERE NULL

(19) UNION → Pasting Result

↓

Combine result set of two / more select statements

→ select c1, c2 FROM tableA
UNION
SELECT c1, c2 FROM tableB

---

① TIME — time
② DATE — Date
③ TIMESTAMP → Date & time
④ TIMESTAMPZ → Date, time & zone

20 SHOW ALL
↓
show all parameters of postgre SQL.

21 SELECT NOW
↓
show the Timestamp information of right now.

22 SELECT TIMEOFDAY()
↓
Timestamp in string / readable format

23 ~~SHOW~~
SELECT CURRENT_TIME
SELECT CURRENT_DATE

24 EXTRACT
↓
obtain sub-component of a date value
- year
- month
- Day
- week
- quarter

EXTRACT (YEAR FROM date_col)

25 AGE
Calculates and returns current age given a timestamp

AGE (date_col)

26 TO_CHAR
convert data types to text

TO_CHAR (date_col, 'mm_dd_yyyy')

27 dow
↓
Extract day of week (start sunday — 0)

28 LENGTH (string)

29 || → concatenate strings

30 upper (string)
↓
making it uppercase.

31 LEFT
↓
select first 'n' characters in a string.
LEFT (string, n)

* SUBQUERY
e.g. students with score > avg class score

```
SELECT student, grade
    FROM test_score
    WHERE grade >
        (SELECT AVG(grade)
         FROM test_scores)
```

32 EXISTS
↓
Used to test existance of rows in a subquery

33 SELF-JOIN
↓
- Table Joined to itself
- useful for comparing values in columns of rows within same table

Data Types

★ Datatypes:

① Boolean
② character
- char
- varchar
- text

③ Numeric
- integer
- floating point

④ Temporal
- date
- time
- timestamp
- Interval

⑤ UUID - Universally Unique Identifiers - Rows

⑥ Array

⑦ JSON

⑧ H-store key-value pair

⑨ Network address and geometric data

NOTE > search for best practices while creating a database

★ Primary Key - a column / number of columns
↓
Unique

★ SERIAL data type

SQL ②

★ Table → constraints

- Primary Key
- Foreign Key.

★ Constraints:

↓

- Rules enforced on data columns on table
- prevent invalid data being entered into a dbase.

- Table constrai.
- column constraints

① NOT NULL
② UNIQUE
③ Prim. Key
④ Foreign Key
⑤ Check - ensures all values in a column satisfy certain condn
⑥ EXCLUSION
⑦ REFERENCES

SYNTAX

① CREATE

```
CREATE TABLE table_name(
    col_name TYPE col_constraint,
    col_name TYPE col_constraint,
    table_constraint table_constraint
) INHERITS existing_table_name
```

② SERIAL — data type

- sequence - a special dbase object that generates sequence of integers
- sequence ⟶ often used as primary key column
- It logs unique integer entries automatically upon insertion

Note: If a row is removed, the column with serial data type will not adjust.

1, 2, 3, 4, 6, 7

It shows row ⑤ was removed

```
CREATE TABLE tab_name(
    col_name SERIAL col_const,
    col_name TYPE col_constraint,
    );
```

* serial should only be used as a PRIMARY KEY for table that its in.

③ REFERENCES - constraint

```
create table table1(
c2 INTEGER REFERENCES
        table2 (c2) )
```

④ INSERT - Rows in a table

```
INSERT INTO table (c1, c2)
        VALUES (v1, v2)
```

* Insert values from an another table

```
INSERT INTO table (c1, c2)
    SELECT c1, c2
        FROM another_table
        WHERE condition
```

Note: SERIAL columns do not need to be provided a value.

⑤ UPDATE - changing values in a column

```
UPDATE table
    SET column1 = value1
        column2 = value2
    WHERE condition
```

* Update (Join)

```
UPDATE tableA
  SET original_col = tableB.new_col
  FROM tableB
  WHERE tableA.id = tableB.id
```

* Reset everything (W/O WHERE)

```
UPDATE account
SET last_login = CURRENT_TIMESTAMP
```

* other column

```
UPDATE account
SET last_login = created_on
RETURNING acc_id, last_login
```

⑥ DELETE — remove ROWs from a table

```
DELETE FROM table
  WHERE row_id = 1
```

* Based on other tables:

```
DELETE FROM tableA
  USING tableB
  WHERE tableA.id=tableB.id
```

* All rows delete

```
DELETE FROM table
```

* RETURNING:

To return rows that were removed.

⑦ ALTER → changes to an existing table struc.

- Add / drop / rename columns
- changing data type of col
- default values to col
- Add CHECK constraints
- Rename table.

```
ALTER TABLE
      table_name
      action

ALTER TABLE
      table_name
      ADD COLUMN
      new_col TYPE

ALTER COLUMN col_name
 SET DEFAULT value
 SET NOT NULL
```

⑧ DROP

- complete removal of a column in a table.
- Also indices & all constraints associated.
- NOTE It will not remove columns used in views, triggers, or stored procedures without add^n CASCADE clause.

```
ALTER TABLE table_name
    DROP COLUMN col_name
```

* Remove all dependancies:

```
ALTER TABLE table_name
DROP COLUMN col_name
 CASCADE
```

* IF EXISTS

```
ALTER TABLE table_name
DROP COLUMN IF EXISTS
 col_name
```

* DROP multiple:

```
ALTER TABLE table_name
DROP COLUMN col_one,
DROP COLUMN col_two
```

⑨ CHECK → Allows to create more customised constraint that adhere to certain cond^n

```
CREATE TABLE example (
  ex_id SERIAL PRIMARY KEY,
  age SMALLINT CHECK (age>21),
  parent_age SMALLINT CHECK (parent
    age > age));
```

✱ Conditional Expressions and Procedures:

(10) CASE → only execute SQL when certain cond^n are met.

```
CASE expression
    WHEN cond^n 1 THEN result1
    WHEN cond^n 2 THEN result2
    ELSE some_other_result
END
```

(11) COALESCE → e.g. replace null with 0.

↓

Accepts an unlimited no. of argument that is not null, if all argu. are NULL then it will return NULL

```
COALESCE (column, 0)
```

(12) CAST → convert from one datatype into other

```
SELECT CAST ('5' AS INTEGER)
```

```
SELECT '5' :: INTEGER
```

e.g.

```
SELECT CAST (date AS
                TIMESTAMP)
FROM table.
```

(13) NULLIF → Takes in 2 inputs and returns NULL if both are equal, otherwise returns first argument passed ↓ (Just to check against zero)

```
NULL (arg1, arg2).
```

(14) VIEW → Instead of performing same query again and again, create a VIEW to call that query

- → a database object of a stored query
- → Can be accessed as a virtual table in PGsql

Note) It does not store the data physically but simply stores a query.

```
CREATE VIEW view_name AS
.....
.....
.....
```

(15) Import / Export → Import .CSV file to already existing table

Note) Import command does not create a table.