Day 2 Contents

- Column Data Type
- Select Statement
- SubQueries
- Constraints
- Union

Column Data Type

- PostgreSQL has a rich set of native data types available to users.
- Users can add new types to PostgreSQL using the CREATE TYPE command.

Numeric data type

Name	Storage Size	Description	Range
smallint	2 bytes	small-range integer	-32768 to +32767
integer	4 bytes	typical choice for integer	-2147483648 to +2147483647
bigint	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
numeric(precision, scale)	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point, number 23.5141 has a precision of 6 and a scale of 4.
smallserial	2 bytes	small autoincrementing integer	1 to 32767
serial	4 bytes	autoincrementing integer	1 to 2147483647
bigserial	8 bytes	large autoincrementing integer	1 to 9223372036854775807

Monetary data type

- The money type stores a currency amount with a fixed fractional precision.
- Input is accepted in a variety of formats, as integer and floating-point literals, as well as typical currency formatting, such as '$1,000.00'

Name	Storage Size	Description	Range
money	8 bytes	currency amount	-92233720368547758.08 to +92233720368547758.07

Character data types

- An attempt to store a longer string will result in an error, unless the excess characters are all spaces, in this case string will be truncated to the maximum
- If the string is shorter than the declared length, values of type character will be space-padded; values of type character varying will simply store the shorter string.

Name	Description
character varying(n), varchar(n)	variable-length with limit
character(n), char(n)	fixed-length, blank padded
text	variable unlimited length

Date/Time data type

- Valid input for the time stamp types consists of the concatenation of a date and a time, followed by an optional time zone, followed by an optional AD or BC.

Name	Storage Size	Description	Examples
timestamp [without time zone]	8 bytes	both date and time (no time zone), From 4713 BC to 294276 AD	1999-01-08 04:05:06 January 8 04:05:06 99 BC
timestamp with time zone	8 bytes	both date and time, with time zone, From 4713 BC to 294276 AD	1999-01-08 04:05:06 -8:00 January 8 04:05:06 1999 PST

Date/Time data type

Name	Description
date	date (no time of day)
time [without time zone]	time of day (no date)
time with time zone	times of day only, with time zone
interval [fields]	time interval, field can be YEAR, MONTH, DAY, HOUR, MINUTE, SECOND

Boolean data type

- Valid literal values for the "true" state are:

TRUE , 't‘ , 'true‘, 'y‘, 'yes‘, 'on‘ or '1‘

- For the "false" state, the following values can be used:

FALSE, 'f‘, 'false‘, 'n‘, 'no‘, 'off‘ or '0'

Name	Storage Size	Description
boolean	1 byte	state of true or false

Enumerated Types

- Enumerated (enum) types are data types that comprise a static, ordered set of values.
- Enum types are created using the CREATE TYPE command, for example:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
SELECT * FROM person WHERE current_mood > ’ok’;
```

Others Data Types

Geometric Types (point, line, circle)

Network Address Types (IP, Mac)

XML Types

JSON Types

Binary Data Types

Composite type

- A *composite type:* it is essentially just a list of field names and their data types.

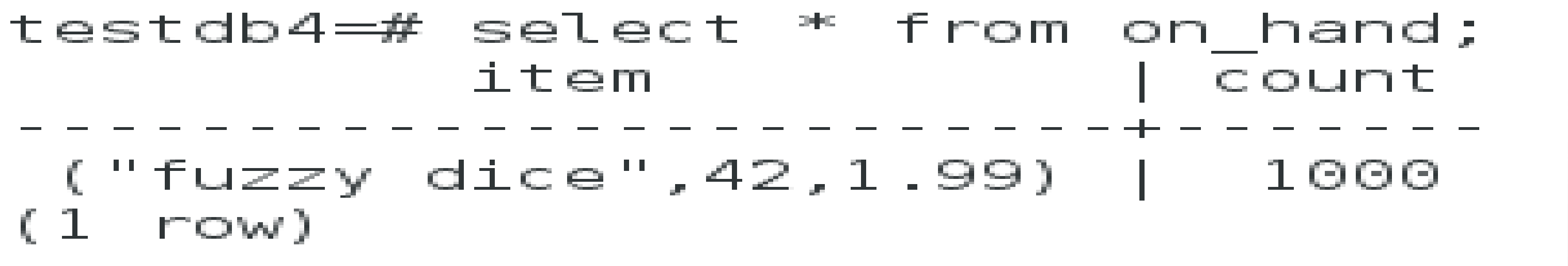
```
CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric(3,2)
);
```

Composite type

```
CREATE TABLE on_hand (item  inventory_item,count integer);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99),

1000);
```



```
testdb4=# select * from on_hand;
           item           | count
--------------------------+-------
 ("fuzzy dice",42,1.99)   |  1000
(1 row)
```

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;

UPDATE on_hand SET item.price = (item).price + 1 WHERE ...;
```

Delete

- The basic syntax of DELETE query with WHERE clause is as follows:

```
DELETE FROM table_name
[ WHERE condition ];
```

- Example:

```
DELETE FROM COMPANY WHERE ID = 2;
```

If you want to DELETE all the records from COMPANY table:

```
DELETE FROM COMPANY;
```

Truncate

- TRUNCATE TABLE command is used to delete complete data from an existing table.
- It has the same effect as an DELETE on each table, but since it does not actually scan the tables, it is faster

```
TRUNCATE TABLE  table_name;
```

Example:

```
TRUNCATE TABLE COMPANY;
```

DROP Table

- Basic syntax of Drop table statement is as follows:

```
DROP TABLE [ IF EXISTS ] name [, ...];

drop table company;
```

Insert

- Basic syntax of INSERT INTO statement is as follows:

```
INSERT INTO TABLE_NAME [(column1, column2,
column3,...columnN)]
VALUES (value1, value2, value3,...valueN);
```

Example:

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY,JOIN_DATE)
VALUES (1, ‘ali', 27, ‘Cairo', 20000.00 ,'2011-07-13');
```

Update

- The basic syntax of UPDATE query with WHERE clause is as follows:

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN =
valueN
[ WHERE condition ];
```

- Example:

```
UPDATE COMPANY SET SALARY = 15000 WHERE ID = 3;
UPDATE COMPANY SET ADDRESS = ‘Giza', SALARY=20000;
```

Select

- PostgreSQL **SELECT** statement is used to fetch the data from a database table
- The basic syntax of SELECT statement is as follows:

```
SELECT column1, column2, columnN FROM table_name;
```

If you want to fetch all the fields available in the table then you can use the following syntax:

```
SELECT * FROM table_name;
```

Select

```
SELECT column1, column2
FROM table1
[ WHERE  conditions ]
[ GROUP BY column ]
[ HAVING conditions ]
[ ORDER BY column ]
[LIMIT no of rows]
```

Where

- WHERE clause is used to specify a condition while fetching the data from single table or joining with multiple tables.
- Example

```
SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
```

Operators

- Operators are used to specify conditions in a PostgreSQL statement and to serve as conjunctions for multiple conditions in a statement.
- It can be classified into:
  - Arithmetic operators
  - Comparison operators
  - Logical operators

Comparison Operators

OP	Description
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!= <>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Arithmetic Operators

OP	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
^	Exponentiation - This gives the exponent value of the right hand operand, 2.0 ^ 3.0 =8
|/	square root, |/ 25.0=5
||/	Cube root, ||/ 27.0 =3
!/	Factorial, 5 ! =120

Logical Operators

OP	Description
AND	The AND operator allows the existence of multiple conditions in a PostgresSQL statement's WHERE clause.
OR	The OR operator is used to combine multiple conditions in a PostgresSQL statement's WHERE clause.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg. NOT EXISTS, NOT BETWEEN, NOT IN etc. This is negate operator.

Others Operators

```
SELECT * FROM COMPANY WHERE AGE IS NULL;

SELECT * FROM COMPANY WHERE AGE NOT IS NULL;

SELECT * FROM COMPANY WHERE AGE IS NOT NULL;


SELECT * FROM COMPANY WHERE AGE IN ( 25, 27 );

SELECT * FROM COMPANY WHERE AGE NOT IN ( 25, 27 );


SELECT * FROM COMPANY WHERE AGE BETWEEN 25 AND 27;

SELECT * FROM Orders WHERE OrderDate BETWEEN #07/04/1996#

AND #07/09/1996#;
```

Like Operators

Statement	Description
WHERE SALARY LIKE '200%'	Finds any values that start with 200
WHERE SALARY LIKE '%2'	Finds any values that end with 2
WHERE SALARY LIKE '_2%3'	Finds any values that have a 2 in the second position and end with a 3

Expressions

- The expression is a combination of one or more values with previous operators.
- Examples:

```
SELECT * FROM COMPANY WHERE SALARY = 10000;

SELECT * FROM COMPANY WHERE (SALARY + 6) = 1005;

SELECT * FROM COMPANY WHERE (SALARY * Age) >= 1005;
```

Aggregation Functions

- compute a single result from a set of input values

`Such as: COUNT(), MAX(), MIN(), AVG(), SUM()`

- Examples:

```
SELECT MAX(salary) FROM COMPANY;

SELECT MIN(salary) FROM COMPANY;

SELECT AVG(SALARY) FROM COMPANY;

SELECT SUM(salary) FROM company;

SELECT COUNT(salary) FROM COMPANY;
```

GROUP BY

The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns. This is done to eliminate redundancy in the output and/or **compute aggregates** that apply to these groups:

```
SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME;
```

HAVING

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

```
SELECT NAME FROM COMPANY GROUP BY name HAVING count(name)
< 2;
```

ORDER BY

- is used to sort the data in ascending or descending order, based on one or more columns:

```
SELECT * FROM COMPANY ORDER BY AGE ASC;      [default]

SELECT * FROM COMPANY ORDER BY NAME DESC;
```

LIMIT

- LIMIT is used to limit the data amount returned by the SELECT statement, OFFSET rows are skipped before starting to count the LIMIT rows that are returned.

```
SELECT * FROM COMPANY LIMIT 4;

SELECT * FROM COMPANY LIMIT 3 OFFSET 2;
```

SELECT DISTINCT

- SELECT DISTINCT is used to eliminate all the duplicate records and fetching only unique records:

```
SELECT DISTINCT name FROM COMPANY;

SELECT DISTINCT name, age FROM COMPANY;
```

Case Expression

The CASE expression is a generic conditional expression, similar to if/else statements

in other programming languages

```
CASE WHEN condition THEN result

    [WHEN condition THEN result]

    [ELSE result]

END



SELECT column1,column2,[CASE Expression] FROM table
```

Case Expression

```
CREATE TABLE test (a integer);

SELECT a,

       CASE WHEN a=1 THEN 'one'

            WHEN a=2 THEN 'two'

            ELSE 'other'

       END

FROM test;
```

CONSTRAINTS

- Constraints are the rules enforced on data columns on table. These are used to prevent invalid data from being entered into the database.

NOT NULL

- By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column.

```
CREATE TABLE COMPANY1(
   ID            INT      NOT NULL,
   NAME          TEXT     NOT NULL,
   AGE           INT      NOT NULL,
   SALARY        INT
);
```

UNIQUE

- UNIQUE Constraint prevents two records from having identical values in a particular column.

```
CREATE TABLE COMPANY3(
   ID              INT PRIMARY KEY,
   NAME           TEXT,
   AGE            INT,
   ADDRESS        CHAR(50) UNIQUE,
   SALARY         float    DEFAULT 50000.00
);
```

FOREIGN KEY

- A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. We say this maintains the referential integrity between two related tables.

```
CREATE TABLE DEPARTMENT1(
   ID INT PRIMARY KEY      NOT NULL,
   DEPT           CHAR(50) NOT NULL,
   EMP_ID         INT      references COMPANY6(ID)
);
```

FOREIGN KEY

- A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. We say this maintains the referential integrity between two related tables.

```
CREATE TABLE DEPARTMENT1(
   ID INT PRIMARY KEY      NOT NULL,
   DEPT              CHAR(50) NOT NULL,
   MANG_ID       INT  references COMPANY7(ID)
   EMP_ID        INT  references COMPANY6(ID) ON DELETE CASCADE
);
```

CHECK

- The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and isn't entered into the table.

```
CREATE TABLE COMPANY5(
   ID             INT     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL CHECK(SALARY > 999)
);
```

ALTER

- Alter is used to change the definition of a table.

```
ALTER TABLE table_name action [, ... ]

ALTER TABLE table_name RENAME TO new_name

ALTER TABLE table_name ADD column data_type

ALTER TABLE table_name DROP column

ALTER TABLE table_name ALTER column SET DATA TYPE data_type

ALTER TABLE table_name RENAME COLUMN column TO new_column

ALTER TABLE table_name ADD CONSTRAINT my_fk FOREIGN KEY (col1)

REFERENCES foreign_table (foreign_field)  ON DELETE CASCADE;
```

UNION

- UNION clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.
- each UNION query must have the same number of columns

```
SELECT EMP_ID, NAME, DEPT FROM COMPANY
UNION
SELECT EMP_ID, NAME, DEPT FROM COMPANY1;
```

Subqueries

- A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
- Subqueries that return more than one row can only be used with multiple value operators, such as IN, NOT IN operator

```
SELECT * FROM COMPANY WHERE ID IN (SELECT ID
    FROM COMPANY_bkp WHERE SALARY > 45000);

INSERT INTO COMPANY_BKP (SELECT * FROM COMPANY);
```