postgres notes

Aug 30, 2023

CONTENTS:

Study notes based on the PostgreSQL docs. Feel free to use.

CHAPTER

ONE

GETTING STARTED

1.1 Introduction

PostgreSQL uses a *client/server* model. Each session consists of:

1. A server process called `postgres`, that:
   - manages database files
   - accepts connections from client applications
   - performs database actions on behalf of clients.

   The server process can handle multiple concurrent clients by *starting a fork for each new client connection*.
2. A client (frontend) application, which could be:
   - a text-oriented tool
   - a graphical application (gui)
   - a web server that accesses the database to display web pages
   - a specialized database maintenance tool.

The client and server *can be on different hosts*, communicating via a TCP/IP network connection. Just ensure that files you intend to use are accessible at both ends.

1.1.1 The Command Line Interface

1. Creating a database

Use the createdb command:

```
$ createdb mydb
```

Database names shoud start with an alphabetic character, and must be <= 63 bytes long. If you don't provide a database name, the current username will be used.

2. Deleting a database

Use the dropdb command:

```
$ dropdb mydb
```

Permanently removes all files related to the database. Can't be undone. Database name must always be specified.

3. Accessing a database

You can use:

- the *PostgreSQL* interactive terminal program, `psql`

```
$ psql mydb
psql (15.3)
Type "help" for help.

mydb=>
```

- a graphical frontend tool e.g. pgAdmin
- a custom application, using available language bindings.

We'll focus on *psql*.

1.2 Basics

SQL (Structured Query Language) is a domain-specific language used to manage / process data stored in relational database management systems (RDBMS) e.g *MySQL*, *PostgreSQL*, *SQLite*. A *relation* is a **table**.

A table is a named collection of **row**s. Several tables can be grouped in a **database**. A collection of databases managed by a single server instance is called a **database cluster**.

Each row has the same set of named **column**s. The order of rows within a table is not guaranteed, but you can sort values for display.

Each column has a *specific data type*, and a *fixed order* in each row.

Spaces, tabs and newlines can be used freely in SQL commands.

`--` introduces single-line comments.

1.2.1 1. Creating a table

Use a `CREATE TABLE` statement with column names and their data types:

```
$ psql mydb
psql (15.3)
Type "help" for help.
mydb=>
```

```
mydb=> CREATE TABLE products (
mydb(>   name             varchar(50),
mydb(>   items_in_stock   int,
mydb(>   price            numeric(7, 2)
mydb(> );
CREATE TABLE
```

1.2.2 2. Populating a table

Use an `INSERT` command:

```
mydb=> INSERT INTO products (name, items_in_stock, price) VALUES ('Apples', 100, 25);
INSERT 0 1
```

You can list columns in any order, with their respective values:

```
mydb=> INSERT INTO products (price, name, items_in_stock) VALUES (10, 'Bananas', 32);
INSERT 0 1
```

You can insert values without specifying columns (not recommended):

```
mydb=> INSERT INTO products VALUES ('Cherries', 74, 2.5);
INSERT 0 1
```

You can also use the `COPY` command to load large amounts of data from *flat-text files* (e.g. txt, csv) into a table.

Tip: The *psql* `\copy` command is more user-friendly when fetching/storing data in a file *accessible to the psql client*:

```
\copy table_name FROM '/path/to/data.csv'
```

1.2.3 3. Querying a table

Use a `SELECT` statement:

```
mydb=> SELECT * FROM products;
   name   | items_in_stock | price
----------+----------------+-------
 Apples   |            100 | 25.00
 Bananas  |             32 | 10.00
 Cherries |             74 |  2.50
(3 rows)
```

`*` is shorthand for "all columns". You can specify columns (recommended):

```
mydb=> SELECT name, price FROM products;
   name   | price
----------+-------
 Apples   | 25.00
 Bananas  | 10.00
```

(continues on next page)

(continued from previous page)

```
 Cherries |  2.50
(3 rows)
```

You can include *expressions*:

```
mydb=> SELECT name, items_in_stock * price AS inventory_value FROM products;
   name   | inventory_value
----------+-----------------
 Apples   |         2500.00
 Bananas  |          320.00
 Cherries |          185.00
(3 rows)
```

You can use a `WHERE` clause to filter results:

```
mydb=> SELECT * FROM products WHERE items_in_stock < 50;
  name   | items_in_stock | price
---------+----------------+-------
 Bananas |             32 | 10.00
(1 row)
```

You can use an `ORDER BY` clause to sort results:

```
mydb=> SELECT * FROM products ORDER BY price;
   name   | items_in_stock | price
----------+----------------+-------
 Cherries |             74 |  2.50
 Bananas  |             32 | 10.00
 Apples   |            100 | 25.00
```

1.2.4 4. Joining tables

A *join query* accesses multiple tables (or multiple instances of the same table) at once:

We'll need another table to experiment with joins:

```
mydb=> CREATE TABLE suppliers (
mydb(>   name                varchar(70),
mydb(>   product_name        varchar(50),
mydb(>   unit_price          numeric(7, 2),
mydb(>   last_delivery_date  date
mydb(> );
CREATE TABLE
mydb=> INSERT INTO suppliers VALUES
mydb->   ('ACME Fruits Ltd', 'Bananas', 8.5, '2023-07-23'),
mydb->   ('Green Thumb Corp.', 'Spinach', 5.95, '2023-07-24'),
mydb->   ('Jolly Grocers', 'Apples', 23.80, '2023-07-24');
INSERT 0 3
```

```
mydb=> SELECT * FROM suppliers;
       name        | product_name | unit_price | last_delivery_date
-------------------+--------------+------------+--------------------
```

(continues on next page)

(continued from previous page)

```
 ACME Fruits Ltd   | Bananas      |       8.50 | 2023-07-23
 Green Thumb Corp. | Spinach      |       5.95 | 2023-07-24
 Jolly Grocers     | Apples       |      23.80 | 2023-07-24
(3 rows)
```

It is good practice to *qualify* column names (e.g table.colname) and use *aliases* to avoid issues with duplicate column names:

```
mydb=> SELECT * FROM products JOIN suppliers ON name = product_name; -- this will fail
ERROR:  column reference "name" is ambiguous
LINE 1: SELECT * FROM products JOIN suppliers ON name = product_name...
```

```
mydb=> SELECT * FROM products p JOIN suppliers s ON p.name = s.product_name;  -- 2 "name
↪" columns
  name   | items_in_stock | price |      name       | product_name | unit_price | last_
↪delivery_date
---------+----------------+-------+-----------------+--------------+------------+--------
↪------------
 Apples  |            100 | 25.00 | Jolly Grocers   | Apples       |      23.80 | 2023-
↪07-24
 Bananas |             32 | 10.00 | ACME Fruits Ltd | Bananas      |       8.50 | 2023-
↪07-23
(2 rows)
```

```
mydb=> SELECT s.name AS supplier_name, p.name AS product_name, p.price,
mydb->        s.unit_price AS purchase_price, p.items_in_stock, s.last_delivery_date
mydb->   FROM products p JOIN suppliers s ON p.name = s.product_name;
  supplier_name  | product_name | price | purchase_price | items_in_stock | last_
↪delivery_date
-----------------+--------------+-------+----------------+----------------+--------------
↪------
 Jolly Grocers   | Apples       | 25.00 |          23.80 |            100 | 2023-07-24
 ACME Fruits Ltd | Bananas      | 10.00 |           8.50 |             32 | 2023-07-23
(2 rows)
```

The default is an *inner join*, which returns only rows that match the join condition. To include all possible results from both tables, we can use a *full outer join*:

```
mydb=> SELECT s.name AS supplier_name, p.name AS product_name, p.price,
mydb->        s.unit_price AS purchase_price, p.items_in_stock, s.last_delivery_date
mydb->   FROM products p FULL OUTER JOIN suppliers s ON p.name = s.product_name
mydb->   ORDER BY supplier_name;
   supplier_name   | product_name | price | purchase_price | items_in_stock | last_
↪delivery_date
-------------------+--------------+-------+----------------+----------------+------------
↪--------
 ACME Fruits Ltd   | Bananas      | 10.00 |           8.50 |             32 | 2023-07-23
 Green Thumb Corp. |              |       |           5.95 |                | 2023-07-24
 Jolly Grocers     | Apples       | 25.00 |          23.80 |            100 | 2023-07-24
                   | Cherries     |  2.50 |                |             74 |
(4 rows)
```

More on joins later.

1.2.5 5. Aggregate functions

Aggregate functions compute a *single result from multiple input rows* e.g. `count`, `sum`, `avg`, `max` and `min`.

```
mydb=> SELECT sum(price * items_in_stock) FROM products AS total_inventory_value;
   sum
---------
 3005.00
(1 row)
```

```
mydb=> SELECT min(price), max(price), avg(price) FROM products;
 min  |  max  |         avg
------+-------+---------------------
 2.50 | 25.00 | 12.5000000000000000
(1 row)
```

To include aggregate functions in `WHERE` clauses, you can use a *subquery*. `WHERE` clauses determine which rows to include, and so are processed before aggregate functions.

```
mydb=> SELECT name, price FROM products WHERE price = min(price);
ERROR:  aggregate functions are not allowed in WHERE
LINE 1: SELECT name, price FROM products WHERE price = min(price);
                                                       ^
```

```
mydb=> SELECT name, price FROM products WHERE price = (SELECT min(price) FROM products);
   name   | price
----------+-------
 Cherries |  2.50
(1 row)
```

Aggregate functions are often used in `GROUP BY` clauses:

```
mydb=> INSERT INTO suppliers (name, product_name, unit_price, last_delivery_date) VALUES
mydb->   ('Planet Farms', 'Apples', 24.10, '2023-07-25'),
mydb->   ('City Merchants', 'Bananas', 9.00, '2023-07-25');
INSERT 0 2
mydb=> SELECT product_name, count(name) AS num_suppliers, min(unit_price) AS min_price,
mydb->        max(unit_price) AS max_price, avg(unit_price) AS avg_price
mydb->  FROM suppliers
mydb->  GROUP BY product_name;
 product_name | num_suppliers | min_price | max_price |      avg_price
--------------+---------------+-----------+-----------+---------------------
 Apples       |             2 |     23.80 |     24.10 | 23.9500000000000000
 Bananas      |             2 |      8.50 |      9.00 |  8.7500000000000000
 Spinach      |             1 |      5.95 |      5.95 |  5.9500000000000000
(3 rows)
```

```
mydb=> SELECT s.product_name, avg(s.unit_price) AS avg_purchase_price, p.price AS sale_
↪price
mydb->   FROM products p JOIN suppliers s ON p.name = s.product_name
mydb->   GROUP BY s.product_name, p.price
mydb->   ORDER BY sale_price DESC;
 product_name | avg_purchase_price  | sale_price
```

(continues on next page)

(continued from previous page)

```
-------------+---------------------+------------
 Apples      | 23.9500000000000000 |      25.00
 Bananas     |  8.7500000000000000 |      10.00
```

You can filter grouped rows with a `HAVING` clause.

```
mydb=> SELECT product_name, avg(unit_price) AS avg_purchase_price
mydb->   FROM suppliers
mydb->   GROUP BY product_name
mydb->   HAVING avg(unit_price) < 10;
 product_name | avg_purchase_price
--------------+--------------------
 Bananas      | 8.7500000000000000
 Spinach      | 5.9500000000000000
(2 rows)
```

Note: The fundamental difference between `WHERE` and `HAVING` is that `WHERE` selects *input rows before grouping & aggregation*, whereas `HAVING` selects *group rows after groups and aggregates are computed*.

`HAVING` clauses usually contain aggregate functions, but this isn't a must. In such cases, `WHERE` clauses would be more efficient; we'd avoid doing grouping and aggregate calculations for all rows that fail the `WHERE` check.

1.2.6 6. Updates

You can update existing rows using the `UPDATE` command:

```
mydb=> UPDATE products SET price = 3 WHERE name = 'Cherries';
UPDATE 1
mydb=> SELECT * FROM products;
   name    | items_in_stock | price
-----------+----------------+-------
 Apples    |            100 | 25.00
 Bananas   |             32 | 10.00
 Cherries  |             74 |  3.00
(3 rows)
```

1.2.7 7. Deletions

You can remove rows using the `DELETE` command:

```
mydb=> DELETE FROM products WHERE name = 'Bananas';
DELETE 1
mydb=> SELECT * FROM products;
   name    | items_in_stock | price
-----------+----------------+-------
 Apples    |            100 | 25.00
 Cherries  |             74 |  3.00
(2 rows)
```

Caution: `DELETE FROM tablename;` will remove all rows. Be specific.

Tip: Start with a `SELECT` query to check the rows that would be selected. Then switch to a `DELETE` command.

You can use `DROP TABLE` to remove a table:

```
mydb=> DROP TABLE products;
DROP TABLE
mydb=> SELECT * FROM products;
ERROR:  relation "products" does not exist
LINE 1: SELECT * FROM products;
                      ^
```

1.3 Intermediate

First, let's refresh the sample data:

```
$ dropdb mydb  # start afresh
$ createdb mydb
$ psql mydb
psql (15.3)
Type "help" for help.

mydb=>
```

```
mydb=> \i sample_tables.sql
BEGIN
CREATE TABLE
INSERT 0 10
CREATE TABLE
INSERT 0 8
CREATE TABLE
INSERT 0 15
COMMIT
```

1.3.1 1. Views

Creating a view over a query *gives it a name* that you can refer to like an ordinary table:

```
mydb=> CREATE VIEW price_info AS
mydb->   SELECT pu.supplier_name, pu.product_name, pr.price AS selling_price,
mydb->          pu.unit_price AS purchase_price, (pr.price - pu.unit_price) AS profit_
↪per_unit
mydb->     FROM purchases pu JOIN products pr ON pu.product_name = pr.name
mydb->     ORDER BY profit_per_unit DESC;
CREATE VIEW
mydb=> SELECT * FROM price_info LIMIT 5;
```

(continues on next page)

(continued from previous page)

```
       supplier_name        | product_name | selling_price | purchase_price | profit_
↪per_unit
----------------------------+--------------+---------------+----------------+-----------
↪------
 Zing Gardens               | Watermelons  |         42.00 |          39.95 |          ␣
↪ 2.05
 City Merchants             | Bananas      |         10.00 |           8.00 |          ␣
↪ 2.00
 Green Thumb Corp.          | Spinach      |          7.50 |           5.95 |          ␣
↪ 1.55
 ACME Fruits Ltd            | Bananas      |         10.00 |           8.50 |          ␣
↪ 1.50
 Village Growers Association | Mangoes      |         30.00 |          28.50 |          ␣
↪ 1.50
(5 rows)
```

Using views is considered *good SQL database design*. You can use views almost anywhere a table can be used. You can build views upon other views.

1.3.2 2. Foreign keys

Foreign keys maintain *referential integrity*, ensuring that you can't insert values in one table that do not have a matching reference in another.

```
mydb=> INSERT INTO purchases VALUES('Planet Farms', 'Coconuts', 10, 15.00, '2023-07-29');
ERROR:  insert or update on table "purchases" violates foreign key constraint "purchases_
↪product_name_fkey"
DETAIL:  Key (product_name)=(Coconuts) is not present in table "products".
```

More on *foreign keys* and other *constraints* later.

1.3.3 3. Transactions

Transactions *bundle multiple steps into a single, all-or-nothing operation*.

A transactional database guarantees that all the updates made by a transaction are *logged in permanent storage* (i.e. on disk) before the transaction is reported complete.

Transactions are *atomic*: from the point of view of other transactions, they either happen completely or not at all. Intermediate states between the steps in a transaction are invisible to other concurrent transactions.

```
mydb=> BEGIN; -- record a purchase and update inventory
BEGIN
mydb=*> INSERT INTO purchases (supplier_name, product_name, units, unit_price, last_
↪delivery_date)
mydb-*>   VALUES ('Zing Gardens', 'Pineapples', 30, 33.75, '2023-07-30');
INSERT 0 1
mydb=*> UPDATE products SET items_in_stock = items_in_stock + 30
mydb-*>   WHERE name = 'Pineapples';
UPDATE 1
mydb=*> COMMIT;
COMMIT
```

You can use the ROLLBACK command to cancel an ongoing transaction:

```
mydb=> BEGIN;
BEGIN
mydb=*> INSERT INTO products (name, items_in_stock, price) VALUES ('Pumpkins', 10, 12.
↪00);
INSERT 0 1
mydb=*> ROLLBACK;
ROLLBACK
mydb=> SELECT * FROM products WHERE name = 'Pumpkins';  -- insert was undone by rollback
 name | items_in_stock | price
------+----------------+-------
(0 rows)
```

You can use the SAVEPOINT command to define *savepoints*. You can then use ROLLBACK TO to roll back to your savepoints as many times as you'll need to. No need to start all over.

```
mydb=> BEGIN;
BEGIN
mydb=*> INSERT INTO products (name, items_in_stock, price) VALUES ('Pumpkins', 10, 12.
↪00);
INSERT 0 1
mydb=*> SELECT * FROM products WHERE name = 'Pumpkins';
   name   | items_in_stock | price
----------+----------------+-------
 Pumpkins |             10 | 12.00
(1 row)

mydb=*> SAVEPOINT added_pumpkins;
SAVEPOINT
mydb=*> UPDATE products SET price = 10 WHERE name = 'Pumpkins';
UPDATE 1
mydb=*> SELECT * FROM products WHERE name = 'Pumpkins';
   name   | items_in_stock | price
----------+----------------+-------
 Pumpkins |             10 | 10.00
(1 row)

mydb=*> ROLLBACK TO added_pumpkins;
ROLLBACK
mydb=*> SELECT * FROM products WHERE name = 'Pumpkins';
   name   | items_in_stock | price
----------+----------------+-------
 Pumpkins |             10 | 12.00
(1 row)
mydb=*> COMMIT;
COMMIT
```

1.3.4 4. Window functions

A window function *performs a calculation across a set of table rows that are somehow related to the current row*.

Whereas aggregate functions group rows into single output rows, the rows in window fuctions retain their separate identities.

A window function call always contains an `OVER` clause, which determines how the rows of the query are *split up for processing by the window function*.

A `PARTITION BY` clause within `OVER` divides the rows into groups.

To compare the prices of products from different suppliers against the average:

```
mydb=> SELECT product_name, supplier_name, unit_price,
mydb->        avg(unit_price) OVER (PARTITION BY product_name) AS avg_price
mydb->   FROM purchases
mydb->   ORDER BY avg_price DESC, unit_price DESC;
 product_name |        supplier_name        | unit_price |      avg_price
--------------+-----------------------------+------------+---------------------
 Watermelons  | Zing Gardens                |      39.95 | 39.95000000000000000
 Pineapples   | Zing Gardens                |      33.75 | 33.75000000000000000
 Pineapples   | Zing Gardens                |      33.75 | 33.75000000000000000
 Mangoes      | Tropical Paradise Ltd       |      29.05 | 28.77500000000000000
 Mangoes      | Village Growers Association |      28.50 | 28.77500000000000000
 Apples       | Planet Farms                |      24.10 | 23.80000000000000000
 Apples       | Jolly Grocers               |      23.80 | 23.80000000000000000
 Apples       | Village Growers Association |      23.50 | 23.80000000000000000
 Bananas      | City Merchants              |       9.00 |  8.50000000000000000
 Bananas      | ACME Fruits Ltd             |       8.50 |  8.50000000000000000
 Bananas      | City Merchants              |       8.00 |  8.50000000000000000
 Spinach      | Green Thumb Corp.           |       5.95 |  5.95000000000000000
 Kiwis        | Tropical Paradise Ltd       |       4.00 |  4.00000000000000000
 Tomatoes     | Village Growers Association |       3.80 |  3.80000000000000000
 Lemons       | Tropical Paradise Ltd       |       3.25 |  3.25000000000000000
 Cherries     | Jolly Grocers               |       2.15 |  2.15000000000000000
(16 rows)
```

You can control the order in which rows are processed by window functions using `ORDER BY` within `OVER`.

```
mydb=> SELECT product_name, supplier_name, unit_price,
mydb->        rank() OVER (PARTITION BY product_name ORDER BY unit_price DESC)
mydb->   FROM purchases;
 product_name |        supplier_name        | unit_price | rank
--------------+-----------------------------+------------+------
 Apples       | Planet Farms                |      24.10 |    1
 Apples       | Jolly Grocers               |      23.80 |    2
 Apples       | Village Growers Association |      23.50 |    3
 Bananas      | City Merchants              |       9.00 |    1
 Bananas      | ACME Fruits Ltd             |       8.50 |    2
 Bananas      | City Merchants              |       8.00 |    3
 Cherries     | Jolly Grocers               |       2.15 |    1
 Kiwis        | Tropical Paradise Ltd       |       4.00 |    1
 Lemons       | Tropical Paradise Ltd       |       3.25 |    1
 Mangoes      | Tropical Paradise Ltd       |      29.05 |    1
 Mangoes      | Village Growers Association |      28.50 |    2
```

(continues on next page)

(continued from previous page)

```
 Pineapples   | Zing Gardens                |      33.75 |    1
 Pineapples   | Zing Gardens                |      33.75 |    1
 Spinach      | Green Thumb Corp.           |       5.95 |    1
 Tomatoes     | Village Growers Association |       3.80 |    1
 Watermelons  | Zing Gardens                |      39.95 |    1
(16 rows)
```

For each row, there's a set of rows within its partition called its *window frame*. By default, including `ORDER BY` limits the frame to "from start to current row (plus any rows equal to current row)":

```
mydb=> SELECT unit_price, sum(unit_price) OVER (ORDER BY unit_price) FROM purchases;
 unit_price |  sum
------------+--------
       2.15 |   2.15
       3.25 |   5.40
       3.80 |   9.20
       4.00 |  13.20
       5.95 |  19.15
       8.00 |  27.15
       8.50 |  35.65
       9.00 |  44.65
      23.50 |  68.15
      23.80 |  91.95
      24.10 | 116.05
      28.50 | 144.55
      29.05 | 173.60
      33.75 | 241.10
      33.75 | 241.10
      39.95 | 281.05
(16 rows)
```

When `PARTITION BY` and `ORDER BY` are omitted, the default frame consists of all the rows in one partition:

```
mydb=> SELECT unit_price, sum(unit_price) OVER () FROM purchases;
 unit_price |  sum
------------+--------
       8.50 | 281.05
       5.95 | 281.05
      23.80 | 281.05
      24.10 | 281.05
       9.00 | 281.05
      39.95 | 281.05
      28.50 | 281.05
       3.25 | 281.05
       4.00 | 281.05
       2.15 | 281.05
      33.75 | 281.05
       8.00 | 281.05
      29.05 | 281.05
       3.80 | 281.05
      23.50 | 281.05
      33.75 | 281.05
(16 rows)
```

Note: Window functions are *only permitted in the SELECT list and the ORDER BY clause* of the query. They are forbidden elsewhere, such as in GROUP BY, HAVING and WHERE; since they logically execute after the processing of these clauses.

Additionally, window functions execute after non-window aggregate functions. This means it is valid to include an aggregate function call in the arguments of a window function, but not vice versa.

A query can have multiple window functions. If the same windowing behaviour is required, you can avoid duplication using a WINDOW clause that is then referenced in OVER:

```
mydb=> SELECT product_name, unit_price, avg(unit_price) OVER w, stddev(unit_price) OVER w
mydb->   FROM purchases
mydb->   WINDOW w AS (PARTITION BY product_name);
 product_name | unit_price |         avg         |         stddev
--------------+------------+---------------------+------------------------
 Apples       |      24.10 | 23.8000000000000000 | 0.3000000000000000000
 Apples       |      23.50 | 23.8000000000000000 | 0.3000000000000000000
 Apples       |      23.80 | 23.8000000000000000 | 0.3000000000000000000
 Bananas      |       8.50 |  8.5000000000000000 | 0.5000000000000000000
 Bananas      |       9.00 |  8.5000000000000000 | 0.5000000000000000000
 Bananas      |       8.00 |  8.5000000000000000 | 0.5000000000000000000
 Cherries     |       2.15 |  2.1500000000000000 |
 Kiwis        |       4.00 |  4.0000000000000000 |
 Lemons       |       3.25 |  3.2500000000000000 |
 Mangoes      |      28.50 | 28.7750000000000000 | 0.38890872965260113842
 Mangoes      |      29.05 | 28.7750000000000000 | 0.38890872965260113842
 Pineapples   |      33.75 | 33.7500000000000000 |                      0
 Pineapples   |      33.75 | 33.7500000000000000 |                      0
 Spinach      |       5.95 |  5.9500000000000000 |
 Tomatoes     |       3.80 |  3.8000000000000000 |
 Watermelons  |      39.95 | 39.9500000000000000 |
(16 rows)
```

1.3.5 5. Inheritance

Inheritance allows a table to derive columns from zero or more parent tables.

```
mydb=> CREATE TABLE exotic_fruits (
mydb(>   relative_size  varchar(12),
mydb(>   shelf_life     interval
mydb(> ) INHERITS (products);
CREATE TABLE
mydb=> INSERT INTO exotic_fruits (name, items_in_stock, price, relative_size, shelf_life)
mydb->   VALUES ('Pomegranates', 25, 32.00, 'small', '2 weeks');
INSERT 0 1
mydb=> SELECT * FROM exotic_fruits;
     name     | items_in_stock | price | relative_size | shelf_life
--------------+----------------+-------+---------------+------------
 Pomegranates |             25 | 32.00 | small         | 14 days
(1 row)
```

A row of *exotic_fruits* inherits all columns (name, items_in_stock and price) from its parent, *products*.

By default, the data from a child table is included in scans of its parents (e.g Pomegranates from *exotic_fruits* automatically appears in scans of *products*):

```
mydb=> SELECT * FROM products;
     name     | items_in_stock | price
--------------+----------------+-------
 Apples       |            100 | 25.00
 Bananas      |             32 | 10.00
 Cherries     |             74 |  3.00
 Kiwis        |             54 |  5.00
 Lemons       |             49 |  4.00
 Mangoes      |             38 | 30.00
 Pineapples   |             26 | 35.00
 Spinach      |             19 |  7.50
 Tomatoes     |             43 |  4.50
 Watermelons  |             22 | 42.00
 Pumpkins     |             10 | 12.00
 Pomegranates |             25 | 32.00
(12 rows)
```

`ONLY` can be used to indicate that a query should be run over only the specified table, and not tables below it in the inheritance hierarchy:

```
mydb=> SELECT * FROM ONLY products;
    name     | items_in_stock | price
-------------+----------------+-------
 Apples      |            100 | 25.00
 Bananas     |             32 | 10.00
 Cherries    |             74 |  3.00
 Kiwis       |             54 |  5.00
 Lemons      |             49 |  4.00
 Mangoes     |             38 | 30.00
 Pineapples  |             26 | 35.00
 Spinach     |             19 |  7.50
 Tomatoes    |             43 |  4.50
 Watermelons |             22 | 42.00
 Pumpkins    |             10 | 12.00
(11 rows)
```

More on inheritance later.

CHAPTER

TWO

SQL SYNTAX

2.1 Lexical Structure

SQL input contains a sequence of *commands*. A command contains a sequence of *tokens*, terminated by a semicolon `;` (or end of input stream). Tokens are usually separated by whitespace (space, tab, newline).

A token can be a:

- key word
- identifier
- quoted identifier
- literal / constant
- special character symbol

Comments not tokens (treated like whitespace).

2.1.1 1. Identifiers and key words

Key words have a *fixed meaning* in the SQL language, e.g. SELECT, UPDATE.

Identifiers are *names of tables, columns, or other database objects*; depending on the command they are used in.

Identifiers and key words:

- Must begin with a *letter* or *underscore*. Subsequent characters can be letters, underscores, digits(0-9) or $ (non-standard).
- Should be less than 63 bytes by default, or they'll be truncated (`NAMEDATALEN` defaults to 64, and the limit is NAMEDATALEN - 1)
- Are *case insensitive*, except for **delimited / quoted identifiers**(enclosed in `""`; can include spaces, ampersands(&) and more).
  - In SQL, unquoted identifiers are folded to uppercase. In PostgreSQL they're folded to lowercase.
  - A convention often used is to write key words in upper case and names in lower case.

```
SELECT col_name FROM table_name;
```

Note: A delimited identifier is always an identifier e.g. `"select"` is a name but `select` is a key word.

2.1.2 2. Constants

2.1 String constants

Arbitrary sequences of characters bounded by single quotes `'...'` e.g. 'Hello world!'.

To include a single-quote character within a string constant, write two adjacent single quotes:

```
mydb=> SELECT 'Jane''s book';
  ?column?
-------------
 Jane's book
(1 row)
```

Two string constants that are only separated by *whitespace* ***and*** *at least one newline* are concatenated.

```
mydb=> SELECT 'some'
mydb-> 'text';
 ?column?
----------
 sometext
(1 row)
```

2.2 String constants with C-style escapes

PostgreSQL extension. Specified by `e'...'` or `E'...'`. \ begins a C-like *backslash escape sequence*:

Backslash Escape Sequence	Interpretation
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	tab
\o, \oo, \ooo (o = 0–7)	octal byte value
\xh, \xhh (h = 0–9, A–F)	hexadecimal byte value
\uxxxx, \Uxxxxxxxx (x = 0–9, A–F)	16 or 32-bit hexadecimal Unicode character value

```
mydb=> SELECT e'some\trandom'
mydb-> 'text\n\nthere';  -- e" required only in first line
      ?column?
--------------------
 some    randomtext+
                   +
 there
(1 row)
```

2.3 String constants with Unicode escapes

PostgreSQL extension. Specified by `u&'...'` or `U&'...'`. Allows specifying arbitrary Unicode characters by code point (4-digit or 6-digit hexadecimal, prefixed with \ or \+ respectively).

```
mydb=> SELECT U&'d\0061t\+000061';
 ?column?
----------
 data
(1 row)
```

2.4 Dollar-quoted string constants

PostgreSQL extension. Specified by `$$...$$` or `$optional_tag$...$optional_tag$`. The optional tag is case-sensitive, and can be nested by choosing a different tag at each nesting level.

Contents are taken literally (no escapes), enhancing readability and eliminating the need to double escape characters.

```
mydb=> SELECT $$Jane's book$$;  -- equivalent to SELECT 'Jane''s book';
  ?column?
-------------
 Jane's book
(1 row)
```

Particularly useful in function definitions in PostgreSQL.

2.5 Bit-string constants

Binary notation only allows 0 and 1 e.g. `B'101`, `b'111'`. Hexadecimal notation is preceeded by `x` or `X` e.g. `x'abc'`.

2.6 Numeric constants

General forms:

```
digits                          123456789
digits.[digits][e[+-]digits]    123.45678e-9
[digits].digits[e[+-]digits]    .78e9
digitse[+-]digits               1234e+56
```

- At least one digit must be before or after the decimal point, if one is used.
- At least one digit must follow the exponent marker (e), if one is present.
- Any leading + or - is not part of the constant; it is an operator applied to the constant.

In most cases, a numeric constant will be automatically coerced to the most appropriate type depending on context.

```
mydb=> SELECT -123.456e-7;
   ?column?
---------------
 -0.0000123456
(1 row)
```

2.7 Constants of other types

A constant of an arbitrary type can be entered using any one of the following notations:

- `CAST ( 'string' AS type )`

```
mydb=> SELECT CAST (b'110' AS int); -- Standard SQL
int4
------
    6
(1 row)
```

- `type 'string'`

```
mydb=> SELECT interval '1 decade';
interval
----------
10 years
(1 row)
```

- `'string'::type`

```
mydb=> SELECT '123'::numeric(5,2);  -- historical PostgreSQL
numeric
---------
  123.00
(1 row)
```

2.1.3 3. Operators

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

Restrictions:

- `--` and `/*` cannot appear anywhere in an operator name: interpreted as start of comment.
- multiple-character operator names cannot end in + or - unless they also contain at least one of

  ```
  ~ ! @ # % ^ & | ` ?
  ```

  e.g. `@-` is valid, but `*-` is not.

  This restriction allows PostgreSQL to parse SQL-compliant queries without requiring spaces between tokens.
- When working with non-SQL-standard operator names, separate adjacent operators with spaces to avoid ambiguity.

2.1.4 4. Special characters

- `$`
  - if followed by digits e.g. `$1`, represents a *positional parameter* in the body of a function definition or a prepared statement
  - can be part of an identifier or a dollar-quoted string constant.
- `()`
  - groups expressions and enforces precedence
  - is required as part of the fixed syntax of particular SQL commands.
- `[]` selects elements of an array.
- `,` separates the elements of a list.
- `;` terminates SQL commands.
- `:`
  - selects "slices" from arrays
  - is used in certain SQL dialects(such as Embedded SQL) to prefix variable names.
- `*`
  - in some contexts denotes *all the fields* of a table row or composite value
  - in aggregate functions, specifies that the aggregate does not require any explicit parameter.
- `.`
  - is used in numeric constants
  - separates schema, table, and column names.

2.1.5 5. Comments

A comment is a sequence of characters beginning with double dashes and extending to the end of the line e.g.

```
-- A standard SQL comment
```

C-style block comments are also allowed:

```
/* Multi-line comment
* with nesting: /* nested block comment */
*/;
```

Comments are removed from the input stream before further syntax analysis, and are effectively replaced by whitespace.

2.1.6 6. Operator precedence

Operator/Element	Associativity	Description
.	left	table/column name separator
::	left	PostgreSQL-style typecast
[]	left	array element selection
+ -	right	unary plus, unary minus
^	left	exponentiation
* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
(any other operator)	left	all other native and user-defined operators
BETWEEN IN LIKE ILIKE SIMILAR		range containment, set membership, string matching
< > = <= >= <>		comparison operators
IS ISNULL NOTNULL		IS TRUE, IS FALSE, IS NULL, IS DISTINCT FROM, etc
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction

Note: Operator precedence rules above apply to user-defined operators that have the same names as the built-in operators: a + defined for some custom type will have similar precedence to the built in +.

When a schema-qualified operator name is used in the OPERATOR syntax, the OPERATOR construct is always taken to have the default precedence for *any other operator* in the table above.

```
mydb=> SELECT 3 * 2 + 4;
 ?column?
----------
       10
(1 row)
```

```
mydb=> SELECT 3 OPERATOR(pg_catalog.*) 2 + 4;  -- * has lower precedence
 ?column?
----------
       18
(1 row)
```

2.2 Calling Functions

We'll use the following function in the examples below:

```
mydb=> CREATE FUNCTION due_for_purchase(top_n int, threshold int DEFAULT 24)
mydb->   RETURNS TABLE (name text, num_items_left int) AS '
mydb'>     SELECT name, items_in_stock
mydb'>       FROM products
mydb'>       WHERE items_in_stock < $2
```

(continues on next page)

(continued from previous page)

```
mydb'>       ORDER BY items_in_stock LIMIT $1
mydb'>     '
mydb->   LANGUAGE SQL IMMUTABLE STRICT;
CREATE FUNCTION
```

2.2.1 1. Positional Notation

Arguments are supplied in the same order as defined in the function declaration:

```
mydb=> SELECT * FROM due_for_purchase(10, 100);  -- get at most 10 products with < 100␣
↪items left
     name     | num_items_left
--------------+----------------
 Pumpkins     |             10
 Spinach      |             19
 Watermelons  |             22
 Pomegranates |             25
 Bananas      |             32
 Mangoes      |             38
 Tomatoes     |             43
 Lemons       |             49
 Kiwis        |             54
 Pineapples   |             56
(10 rows)
```

Optional arguments (those with default values) can only be omitted from *right to left*:

```
mydb=> SELECT * FROM due_for_purchase(10);  -- using default threshold (24)
    name     | num_items_left
-------------+----------------
 Pumpkins    |             10
 Spinach     |             19
 Watermelons |             22
(3 rows)
```

2.2.2 2. Named Notation

Arguments are supplied as `arg_name => arg_value`, and in any order:

```
mydb=> SELECT * FROM due_for_purchase(threshold => 50, top_n => 5);
     name     | num_items_left
--------------+----------------
 Pumpkins     |             10
 Spinach      |             19
 Watermelons  |             22
 Pomegranates |             25
 Bananas      |             32
(5 rows)
```

Optional arguments can be omitted in any order.

An older syntax based on := is supported for backward compatibility:

```
mydb=> SELECT * FROM due_for_purchase(top_n := 2);  -- using default threshold (24)
   name    | num_items_left
-----------+----------------
 Pumpkins  |             10
 Spinach   |             19
(2 rows)
```

2.2.3 3. Mixed Notation

Combines positional and named notation. Named arguments cannot precede positional ones.

```
mydb=> SELECT * FROM due_for_purchase(5, threshold => 60);
     name     | num_items_left
--------------+----------------
 Pumpkins     |             10
 Spinach      |             19
 Watermelons  |             22
 Pomegranates |             25
 Bananas      |             32
(5 rows)
```

2.3 Value Expressions

2.3.1 Column References

A column can be referenced in the form:

```
[correlation.]column_name
```

where *correlation* could be:

- a table name (possibly qualified with a schema name)
- an alias for a table
- omitted if `column_name` is unique across all tables in the query.

```
mydb=> SELECT p.product_name, p.unit_price AS buying_price,
mydb->        public.products.price AS selling_price, items_in_stock, last_delivery_date
mydb->   FROM purchases p JOIN products ON p.product_name = products.name
mydb->   LIMIT 5;
 product_name | buying_price | selling_price | items_in_stock | last_delivery_date
--------------+--------------+---------------+----------------+--------------------
 Apples       |        23.80 |         25.00 |            100 | 2023-07-24
 Apples       |        24.10 |         25.00 |            100 | 2023-07-25
 Apples       |        23.50 |         25.00 |            100 | 2023-07-28
 Bananas      |         8.50 |         10.00 |             32 | 2023-07-23
 Bananas      |         9.00 |         10.00 |             32 | 2023-07-25
(5 rows)
```

2.3.2 Positional Parameters

Used in function definitions and prepared queries to reference values supplied externally to an SQL statement.

Are of the form `$number` e.g.

```
mydb=> CREATE FUNCTION sum_modulo_n(a int, b int, n int DEFAULT 10) RETURNS int AS
mydb->   'SELECT ($1 + $2) % $3' LANGUAGE SQL;  -- function body with positional params
mydb=> SELECT sum_modulo_n(1, 2), sum_modulo_n(1, 2, 3);
 sum_modulo_n | sum_modulo_n
--------------+--------------
            3 |            0
(1 row)
```

2.3.3 Subscripts

Subscripts select element(s) from arrays:

- `expression[subscript]` for a specific element
- `expression[lower_subscript:upper_subscript]` for an *array slice* (multiple adjacent elements)

```
mydb=> SELECT products AS whole_array, products[1] AS first_element,
mydb->        products[2:3] AS second_and_third
mydb->   FROM suppliers
mydb->   WHERE array_length(products, 1) > 2;
        whole_array         | first_element |  second_and_third
----------------------------+---------------+--------------------
 {Kiwis,Lemons,Mangoes}     | Kiwis         | {Lemons,Mangoes}
 {Apples,Mangoes,Tomatoes}  | Apples        | {Mangoes,Tomatoes}
(2 rows)
```

`subscript` is rounded off to the nearest integer:

```
mydb=> SELECT products, products[0.75] AS sub_0_75, products[1] AS sub_1,
mydb->        products[1.25] AS sub_1_25
mydb->   FROM suppliers
mydb->   WHERE array_length(products, 1) > 1;
         products           |  sub_0_75  |   sub_1    |  sub_1_25
----------------------------+------------+------------+------------
 {Apples,Cherries}          | Apples     | Apples     | Apples
 {Apples,Coconuts}          | Apples     | Apples     | Apples
 {Kiwis,Lemons,Mangoes}     | Kiwis      | Kiwis      | Kiwis
 {Apples,Mangoes,Tomatoes}  | Apples     | Apples     | Apples
 {Pineapples,Watermelons}   | Pineapples | Pineapples | Pineapples
(5 rows)
```

The array `expression` *should be in parenthesis* e.g `(expression)[1]`(but this can be omitted for column references or positional parameters).

Multiple subscripts can be concatenated if the array is multi-dimensional e.g `expression[sub1][sub2]`.

2.3.4 Field Selection

For an `expression` that yields a composite type (row), a specific field of the row can be extracted as `expression.field_name`.

`expression` should be in parenthesis (but this can be omitted for table references or positional parameters):

```
mydb=> SELECT due_for_purchase(3);  -- func returns rows with fields (name, num_items_
↪left)
 due_for_purchase
------------------
 (Pumpkins,10)
 (Spinach,19)
 (Watermelons,22)
(3 rows)
```

```
mydb=> SELECT due_for_purchase(3).name;  -- no parenthesis
ERROR:  syntax error at or near "."
LINE 1: SELECT due_for_purchase(3).name;
                                   ^
```

```
mydb=> SELECT (due_for_purchase(3)).name;  -- with parenthesis
    name
-------------
 Pumpkins
 Spinach
 Watermelons
(3 rows)
```

2.3.5 Operator Invocations

There are two possible syntaxes:

- `expression operator expression` (binary infix operator):

```
mydb=> SELECT 2 + 3;
?column?
----------
        5
(1 row)
```

```
mydb=> SELECT 2 OPERATOR(pg_catalog.+) 3;  --schema-qualified operator name
?column?
----------
        5
(1 row)
```

- `operator expression` (unary prefix operator):

```
mydb=> SELECT -3;
?column?
----------
```

(continues on next page)

(continued from previous page)

```
      -3
(1 row)
```

CHAPTER

THREE

DATA DEFINITION

3.1 Table Basics

Tables consist of columns and rows.

The *number* and *order* of columns is *fixed*. Each column has a *data type* which constrains the set of possible values assignable to it, and enables computation e.g math for numbers and concatenation for strings.

Depending on column type, a table can have as many as 250 - 1600 columns.

The number of rows reflects the amount of data stored, and *order is not guaranteed*. A table can have duplicate rows.

3.1.1 Creating Tables

Use the `CREATE TABLE` command. Specify a table name, column names and column data types.

```
mydb=> CREATE TABLE customers (
mydb(>    first_name   text,
mydb(>    last_name    text,
mydb(>    address      text
mydb(> );
CREATE TABLE
```

3.1.2 Deleting Tables

Use the `DROP TABLE` command:

```
mydb=> DROP TABLE customers;
DROP TABLE
```

3.2 Default Values

A column can be assigned a default value to be used when a new row doesn't specify a value for it. In a table definition, default values are listed after the column data type.

The default value can be an expression which will be evaluated whenever the new row is inserted e.g `CURRENT_DATE`.

```
mydb=> CREATE TABLE animal_products (
mydb(>   name           text,
mydb(>   perishable     bool DEFAULT 'true',
mydb(>   date_produced  date DEFAULT CURRENT_DATE
mydb(> );
CREATE TABLE
mydb=> INSERT INTO animal_products (name) VALUES ('Mutton');
INSERT 0 1
mydb=> INSERT INTO animal_products (name, perishable, date_produced)
mydb->   VALUES ('Leather', 'false', '2023-07-15');
INSERT 0 1
```

```
mydb=> SELECT * FROM animal_products;  -- 'Mutton' row was filled with defaults
  name   | perishable | date_produced
---------+------------+---------------
 Mutton  | t          | 2023-08-03
 Leather | f          | 2023-07-15
(2 rows)
```

If no default value is explicitly declared, the default value is `NULL`.

```
mydb=> INSERT INTO animal_products (perishable) VALUES ('f');  -- use default name␣
↪(NULL) & date
INSERT 0 1
mydb=> SELECT * FROM animal_products;
  name   | perishable | date_produced
---------+------------+---------------
 Mutton  | t          | 2023-08-03
 Leather | f          | 2023-07-15
         | f          | 2023-08-03
(3 rows)

mydb=> DELETE FROM animal_products WHERE name IS NULL;
DELETE 1
```

3.3 Generated Columns

A generated column is computed from other columns. There are 2 kinds:

1. **Stored Generated Columns:**
    - Computed when they are *inserted / updated*.
    - Occupy storage like normal columns.
2. **Virtual Generated Columns:**

- Computed when they are *read*.
- Occupy no storage.
- Not yet implemented in *postgres*.

To create a generated column, use the `GENERATED ALWAYS AS` clause in `CREATE TABLE`. The keyword `STORED` must be specified to choose the stored kind of generated column:

```
mydb=> CREATE TABLE roadtrip (
mydb(>   start_location   text,
mydb(>   start_time       timestamp,
mydb(>   end_location     text,
mydb(>   end_time         timestamp,
mydb(>   duration         interval GENERATED ALWAYS AS (end_time - start_time) STORED
mydb(> );
CREATE TABLE
```

```
mydb=> INSERT INTO roadtrip (start_location, start_time, end_location, end_time)
mydb->   VALUES ('Malindi', '2023-08-01 09:12', 'Changamwe', '2023-08-01 10:55');
INSERT 0 1
```

A generated column can't be written to directly. But the keyword `DEFAULT` may be specified in `INSERT` and `UPDATE` commands:

```
mydb=> INSERT INTO roadtrip (start_location, start_time, end_location, end_time,␣
↪duration)
mydb->   VALUES ('Nairobi', '2023-08-04 06:04', 'Naivasha', '2023 08-04 07:19', '1:15:00
↪');
ERROR:  cannot insert a non-DEFAULT value into column "duration"
DETAIL:  Column "duration" is a generated column.
```

```
mydb=> INSERT INTO roadtrip (start_location, start_time, end_location, end_time,␣
↪duration)
  VALUES ('Nairobi', '2023-08-04 06:04', 'Naivasha', '2023 08-04 07:19', DEFAULT);
INSERT 0 1
mydb=> SELECT * FROM roadtrip;
 start_location |     start_time      | end_location |      end_time       | duration
----------------+---------------------+--------------+---------------------+----------
 Malindi        | 2023-08-01 09:12:00 | Changamwe    | 2023-08-01 10:55:00 | 01:43:00
 Nairobi        | 2023-08-04 06:04:00 | Naivasha     | 2023-08-04 07:19:00 | 01:15:00
(2 rows)
```

Column default	Generated column
Evaluated once, on insert	Updated whenever the row changes
May not refer to other columns	Usually refers to other columns
Can use volatile functions e.g. `random()`	Cannot use volatile functions

3.3.1 Rules

- The generation expression can only use *immutable functions*, and cannot use subqueries or reference anything other than the current row.
- A generation expression cannot reference another generated column.
- A generation expression cannot reference a system column, except `tableoid`.
- A generated column cannot have a column default or identity definition.
- A generated column cannot be part of a partition key.
- Foreign tables can have generated columns (`CREATE FOREIGN TABLE`).
- For inheritance:
  - If a parent column is a generated column, a child column must also be a generated column using the same expression. In the definition of the child column, leave off the GENERATED clause, as it will be copied from the parent.
  - In case of multiple inheritance, if one parent column is a generated column, then all parent columns must be generated columns and with the same expression.
  - If a parent column is not a generated column, a child column may be defined to be a generated column or not.

Generated columns *maintain access privileges separately* from their underlying base columns. You can arrange for a particular role to only read from a generated column but not from the underlying base columns.

Generated columns are updated after `BEFORE` triggers have run. Changes made to base columns in a `BEFORE` trigger will be reflected in generated columns. However, it is not allowed to access generated columns in `BEFORE` triggers.

3.4 Constraints

Constraints enable you to set restrictions on the data storable in table columns (i.e. other than data type). If data to be entered violates a constraint, an error is raised (even if the value is a default).

Constraints can be written in forms:

- **Column constraints**: follow a column definition's data type, and apply to it alone e.g:

```
CREATE TABLE table_name (
  column_name     data_type column_constraint(s),
  ...
);
```

- **Table constraints**: written separately from column definitions e.g:

```
CREATE TABLE table_name (
  column_name    data_type,
  ...
  table_constraint,
  ...
);
```

  Table constraints and column definitions can be written in any order.

Some column constraints can be written as table constraints:

```
CREATE TABLE table_name (
  col1    data_type some_constraint,  --column constraint
  col2    data_type,
  some_constraint (col2)  -- table constraint
);
```

To name a constraint, use the key word `CONSTRAINT`, followed by a name, followed by the constraint definition. Otherwise, the system chooses a name for you.

3.4.1 1. Check Constraints

Specify that the values in a column *must satisfy a Boolean expression*. Consist of the `CHECK` key word, and an expression in `()`:

```
mydb=> CREATE TABLE beverages (
mydb(>   name           text,
mydb(>   price          numeric(5,2) CHECK (price > 0),
mydb(>   serving_temp   text
mydb(> );
CREATE TABLE
```

Values are only included if the check expression evaluates to `true` or `null`:

```
mydb=> INSERT INTO beverages (name, price, serving_temp) VALUES ('Water', 0, 'cool');
ERROR:  new row for relation "beverages" violates check constraint "beverages_price_check
↪"
DETAIL:  Failing row contains (Water, 0.00, cool).
mydb=> INSERT INTO beverages (name, serving_temp) VALUES ('Water', 'cool');  -- null␣
↪price (default)
INSERT 0 1
mydb=> INSERT INTO beverages (name, price, serving_temp) VALUES ('Tea', 9.99, 'hot');
INSERT 0 1
mydb=> SELECT * FROM beverages;
 name  | price | serving_temp
-------+-------+--------------
 Water |       | cool
 Tea   |  9.99 | hot
(2 rows)
```

You can name the check constraint to easily reference it and to clarify error messages.

```
mydb=> DROP TABLE beverages;
DROP TABLE
mydb=> CREATE TABLE beverages (
mydb(>   name           text,
mydb(>   price          numeric(5,2) CONSTRAINT positive_price CHECK (price > 0),
mydb(>   serving_temp   text
mydb(> );
CREATE TABLE
mydb=> INSERT INTO beverages (name, price, serving_temp) VALUES ('Water', 0, 'cool');
ERROR:  new row for relation "beverages" violates check constraint "positive_price"
DETAIL:  Failing row contains (Water, 0.00, cool).
```

A check constraint can refer to multiple columns when written as a table constraint:

```
mydb=> DROP TABLE beverages;
DROP TABLE
mydb=> CREATE TABLE beverages (
mydb(>   name           text,
mydb(>   price          numeric(5,2),
mydb(>   serving_temp   text,
mydb(>   CONSTRAINT valid_beverage CHECK (price > 0
mydb(>                                    AND serving_temp IN ('cold', 'cool', 'room',
↪'warm', 'hot'))
mydb(> );
CREATE TABLE
mydb=> INSERT INTO beverages (name, price, serving_temp) VALUES ('Water', 0, 'cool');
ERROR:  new row for relation "beverages" violates check constraint "valid_beverage"
DETAIL:  Failing row contains (Water, 0.00, cool).
mydb=> INSERT INTO beverages (name, price, serving_temp) VALUES ('Water', 1, 'icy');
ERROR:  new row for relation "beverages" violates check constraint "valid_beverage"
DETAIL:  Failing row contains (Water, 1.00, icy).
mydb=> INSERT INTO beverages (name, price, serving_temp) VALUES ('Water', 1, 'room');
INSERT 0 1
```

Caution: *PostgreSQL* assumes that `CHECK` constraints' conditions are *immutable*. In case of changes, drop the constraint (using `ALTER TABLE`) and then re-add it to re-check it against all rows.

3.4.2 2. Not-Null Constraints

Specify that a column must not assume the `null` value. Always written as column constraints.

```
mydb=> DROP TABLE beverages;
DROP TABLE
mydb=> CREATE TABLE beverages (
mydb(>   name           text NOT NULL,
mydb(>   price          numeric(5,2) NOT NULL,
mydb(>   serving_temp   text
mydb(> );
CREATE TABLE
mydb=> INSERT INTO beverages (name, serving_temp) VALUES ('Milk', 'warm');
ERROR:  null value in column "price" of relation "beverages" violates not-null constraint
DETAIL:  Failing row contains (Milk, null, warm).
mydb=> INSERT INTO beverages (name, price) VALUES ('Milk', 7.50);  -- NULL serving_temp␣
↪not restricted
INSERT 0 1
mydb=> SELECT * FROM beverages;
 name | price | serving_temp
------+-------+--------------
 Milk |  7.50 |
(1 row)
```

Tip: The `NOT NULL` syntax in the example above doesn't support naming. If you must name a not-null constraint,

specify it as `CONSTRAINT constraint_name CHECK (column_name IS NOT NULL)`.

3.4.3 3. Unique Constraints

Ensure that the data contained in a column (or group of columns) is unique among all rows in the table.

```
mydb=> DROP TABLE beverages;
DROP TABLE
mydb=> CREATE TABLE beverages (
mydb(>   name            text UNIQUE,
mydb(>   price           numeric(5,2),
mydb(>   serving_temp    text
mydb(> );
CREATE TABLE
mydb=> INSERT INTO beverages (name, price, serving_temp) VALUES ('Milk', 7.50, 'warm');
INSERT 0 1
mydb=> INSERT INTO beverages (name, price, serving_temp) VALUES ('Milk', 7.50, 'hot');
ERROR:  duplicate key value violates unique constraint "beverages_name_key"
DETAIL:  Key (name)=(Milk) already exists.
```

To define a unique constraint for a group of columns, write it as a table constraint with the column names separated by commas e.g:

```
mydb=> DROP TABLE beverages;
DROP TABLE
mydb=> CREATE TABLE beverages (
mydb(>   name            text,
mydb(>   price           numeric(5,2),
mydb(>   serving_temp    text,
mydb(>   UNIQUE (name, serving_temp)
mydb(> );
CREATE TABLE
mydb=> INSERT INTO beverages (name, price, serving_temp) VALUES ('Milk', 7.50, 'hot');
INSERT 0 1
mydb=> INSERT INTO beverages (name, price, serving_temp) VALUES ('Milk', 6.50, 'hot');
ERROR:  duplicate key value violates unique constraint "beverages_name_serving_temp_key"
DETAIL:  Key (name, serving_temp)=(Milk, hot) already exists.
```

Adding a unique constraint will automatically create a unique *B-tree index* on the column(s) listed in the constraint.

> **Caution:** Since null values are by default considered unequal, it is possible to store duplicate rows that contain a null value in at least one of the constrained columns. Adding a `NULLS NOT DISTINCT` clause or `NOT NULL` constraint can fix this.
>
> Keep in mind that some platforms might implement unique constraints' null treatment differently.

3.4.4 4. Primary Keys

Indicate that a column (or group of columns) can be used as a *unique identifier* for rows in a table (*unique* and *not null*).

A table can have only one primary key. Relational database theory dictates that every table must have a primary key.

```
mydb=> DROP TABLE beverages;
DROP TABLE
mydb=> CREATE TABLE beverages (
mydb(>   name           text,
mydb(>   price          numeric(5,2),
mydb(>   serving_temp   text,
mydb(>   PRIMARY KEY (name, serving_temp)
mydb(> );
CREATE TABLE
mydb=> INSERT INTO beverages (name, price, serving_temp) VALUES ('Lemonade', 5, 'cool');
INSERT 0 1
mydb=> INSERT INTO beverages (name, price, serving_temp) VALUES ('Lemonade', 5.75, 'cool
↪');
ERROR:  duplicate key value violates unique constraint "beverages_pkey"
DETAIL:  Key (name, serving_temp)=(Lemonade, cool) already exists.
mydb=> INSERT INTO beverages (name, price) VALUES ('Lemonade', 5.75);
ERROR:  null value in column "serving_temp" of relation "beverages" violates not-null␣
↪constraint
DETAIL:  Failing row contains (Lemonade, 5.75, null).
```

Adding a primary key will automatically create a unique *B-tree index* on the column(s) listed in the primary key, and will force the column(s) to be marked `NOT NULL`.

A primary key defines the default target column(s) for foreign keys referencing its table.

3.4.5 5. Foreign Keys

Specify that the values in a column (or group of columns) must match the values appearing in some row of another table (maintain *referential integrity* between two related tables).

Extending the example from "Primary Keys" section above:

```
mydb=> CREATE TABLE beverage_sales (
mydb(>   transaction_id     serial PRIMARY KEY,
mydb(>   transaction_time   timestamp DEFAULT CURRENT_TIMESTAMP,
mydb(>   beverage           text,
mydb(>   serving_temp       text,
mydb(>   price              numeric(5,2),
mydb(>   FOREIGN KEY (beverage, serving_temp) REFERENCES beverages (name, serving_temp)
mydb(> );
CREATE TABLE
mydb=> SELECT * FROM beverages;
   name   | price | serving_temp
----------+-------+--------------
 Lemonade |  5.00 | cool
(1 row)

mydb=> INSERT INTO beverage_sales (beverage, serving_temp, price)
```

(continues on next page)

(continued from previous page)

```
mydb->   VALUES ('Lemonade', 'cool', 5.00);
INSERT 0 1
mydb=> INSERT INTO beverage_sales (beverage, serving_temp, price)
mydb->   VALUES ('Lemonade', 'cold', 6.00);
ERROR:  insert or update on table "beverage_sales" violates foreign key constraint
↪"beverage_sales_beverage_serving_temp_fkey"
DETAIL:  Key (beverage, serving_temp)=(Lemonade, cold) is not present in table "beverages
↪"
mydb=> INSERT INTO beverage_sales (beverage, serving_temp, price)
mydb->   VALUES ('Lemonade', DEFAULT, 6.00);  -- Null values might sneak in if not␣
↪constrained
INSERT 0 1
mydb=> SELECT * FROM beverage_sales;
 transaction_id |      transaction_time      | beverage | serving_temp | price
----------------+----------------------------+----------+--------------+-------
              1 | 2023-08-05 10:53:21.48406  | Lemonade | cool         |  5.00
              3 | 2023-08-05 11:01:08.471428 | Lemonade |              |  6.00
(2 rows)
```

We say the *beverage_sales* table is the *referencing* table, and the *beverages* table is the *referenced* table.

You can also define foreign key constraints as column constraints e.g. `column_name data_type REFERENCES other_table (other_table_col)`.

A foreign key must reference columns that are either a primary key or form a unique constraint. In the absence of a column list in a foreign key declaration, the primary key of the referenced table is used as the referenced column(s).

A table can have more than one foreign key constraint, to implement *many-to-many* relationships.

A foreign key that references its own table is called a **self-referential foreign key**. Useful in some cases e.g. to make table rows represent nodes of a tree structure.

You can specify what action to take when an item in a referenced row has to be removed (`ON DELETE`) or changed (`ON UPDATE`):

- `NO ACTION`: Produce an error indicating that the deletion or update would create a foreign key constraint violation.
- `RESTRICT`: Just like NO ACTION, but can't be deferred (is checked immediately).
- `CASCADE`: Delete any rows referencing the deleted row, or update the values of the referencing column(s) to the new values of the referenced columns, respectively.
- `SET NULL`: Set all of the referencing columns (or a specified subset, only for ON DELETE) to null.
- `SET DEFAULT`: Set all of the referencing columns (or a specified subset, only for ON DELETE) to their default values.

e.g.

```
CREATE TABLE order_items (
    product_no  integer REFERENCES products ON DELETE RESTRICT,
    order_id    integer REFERENCES orders ON UPDATE CASCADE,
    quantity    integer,
    PRIMARY KEY (product_no, order_id)
);
```

Tip: If referenced column(s) are changed frequently, it's recommended to add an index to them so that referential actions associated with the foreign key constraint can be performed more efficiently.

3.4.6 6. Exclusion Constraints

Ensure that if any two rows are compared on the specified columns or expressions using the specified operators, at least one of these operator comparisons will return `false` or `null`.

Adding an exclusion constraint will automatically create an index of the type specified in the constraint declaration.

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)  -- no overlapping cirlces
);
```

3.5 System Columns

Every table has several system columns that are implicitly defined by the system:

- **tableoid:** The OID of the table containing this row.
  - Handy for queries that select from *partitioned tables* or *inheritance heirachies* (tells which particular table a row came from)
  - Can be joined with the `oid` column of `pg_class` to obtain the table name.
- **xmin:** The identity (transaction ID) of the inserting transaction for this row version.
- **cmin:** The command identifier (starting at zero) within the inserting transaction.
- **xmax:** The identity (transaction id) of the deleting transaction, or zero for an undeleted row version. If non-zero in a visible row version, this signifies an uncommitted or rolled-back deleting transaction.
- **cmax:** The command identifier within the deleting transaction, or zero.
- **ctid:** The physical location of the row within its table. Changes if a row is updated or moved by `VACUUM FULL`.

```
mydb=> CREATE TABLE alphabet (letter char(1));
CREATE TABLE
mydb=> INSERT INTO alphabet VALUES ('a'), ('b');
INSERT 0 2
mydb=> INSERT INTO alphabet VALUES ('c'), ('d');
INSERT 0 2
mydb=> SELECT letter, tableoid, xmin, cmin, xmax, cmax, ctid FROM alphabet;
 letter | tableoid | xmin | cmin | xmax | cmax | ctid
--------+----------+------+------+------+------+-------
 a      |    17026 | 1038 |    0 |    0 |    0 | (0,1)
 b      |    17026 | 1038 |    0 |    0 |    0 | (0,2)
 c      |    17026 | 1039 |    0 |    0 |    0 | (0,3)
 d      |    17026 | 1039 |    0 |    0 |    0 | (0,4)
(4 rows)
```

Transaction IDs are 32-bit quantities. Uniqueness is not guaranteed for over a billion transactions.

Command identifiers are also 32-bit quantities, limiting each transaction to 2^32 (4,294,967,296) SQL commands. Only commands that actually modify the database contents will consume a command identifier.

3.6 Modifying tables

Use the `ALTER TABLE` command.

We'll use the following table in the examples below:

```
mydb=> DROP TABLE beverages;
DROP TABLE
mydb=> CREATE TABLE beverages (
mydb(>   name           text,
mydb(>   price          numeric(5,2),
mydb(>   serving_temp   text
mydb(> );
CREATE TABLE
mydb=> INSERT INTO beverages VALUES ('Tea', 9.99, 'hot'), ('Lemonade', 5.50, 'cool');
INSERT 0 2
mydb=> SELECT * FROM beverages;
   name    | price | serving_temp
-----------+-------+--------------
 Tea       |  9.99 | hot
 Lemonade |  5.50 | cool
(2 rows)
```

3.6.1 1. Adding a Column

```
mydb=> ALTER TABLE beverages ADD COLUMN serving_quantity_ml integer DEFAULT 350;
ALTER TABLE
mydb=> SELECT * FROM beverages;
   name    | price | serving_temp | serving_quantity_ml
-----------+-------+--------------+---------------------
 Tea       |  9.99 | hot          |                 350
 Lemonade |  5.50 | cool         |                 350
(2 rows)
```

The new column is initially filled with `null` or whatever value is given in the `DEFAULT` clause.

You can include constraints and other options in the column description (just like in `CREATE TABLE`).

3.6.2 2. Removing a Column

```
mydb=> ALTER TABLE beverages DROP COLUMN serving_quantity_ml;
ALTER TABLE
mydb=> SELECT * FROM beverages;
   name    | price | serving_temp
-----------+-------+--------------
 Tea       |  9.99 | hot
```

(continues on next page)

(continued from previous page)

```
 Lemonade |  5.50 | cool
(2 rows)
```

The column's data and constraints disappear. If the column is referenced by a foreign key constraint in another table, an error is raised unless you add a `CASCADE` clause.

3.6.3 3. Adding a Constraint

Use the table constraint syntax:

```
mydb=> ALTER TABLE beverages ADD UNIQUE (name, serving_temp);
ALTER TABLE
mydb=> INSERT INTO beverages VALUES ('Tea', 9.50, 'hot');
ERROR:  duplicate key value violates unique constraint "beverages_name_serving_temp_key"
DETAIL:  Key (name, serving_temp)=(Tea, hot) already exists.
```

To add a not-null constraint (can't be written as a table constraint) use:

```
mydb=> ALTER TABLE beverages ALTER COLUMN name SET NOT NULL;
ALTER TABLE
mydb=> INSERT INTO beverages (price, serving_temp) VALUES (9.50, 'hot');
ERROR:  null value in column "name" of relation "beverages" violates not-null constraint
DETAIL:  Failing row contains (null, 9.50, hot).
```

The constraint will be checked immediately, so the table data must satisfy it before it can be added.

3.6.4 4. Removing a Constraint

You'll need to know the constraint's name. The *psql* command `\d tablename` can help.

```
mydb=> \d beverages
                  Table "public.beverages"
    Column    |     Type      | Collation | Nullable | Default
--------------+---------------+-----------+----------+---------
 name         | text          |           | not null |
 price        | numeric(5,2)  |           |          |
 serving_temp | text          |           |          |
Indexes:
    "beverages_name_serving_temp_key" UNIQUE CONSTRAINT, btree (name, serving_temp)
```

```
mydb=> ALTER TABLE beverages DROP constraint beverages_name_serving_temp_key;
ALTER TABLE
mydb=> INSERT INTO beverages VALUES ('Tea', 9.50, 'hot');
INSERT 0 1
mydb=> SELECT * FROM beverages;  -- duplicate ('Tea', 'hot') pairs
   name    | price | serving_temp
-----------+-------+--------------
 Tea       |  9.99 | hot
 Lemonade  |  5.50 | cool
 Tea       |  9.50 | hot
```

Add `CASCADE` to drop a constraint that something else depends on.

To drop a `NOT NULL` constraint (can't be named) use:

```
mydb=> ALTER TABLE beverages ALTER COLUMN name DROP NOT NULL;
ALTER TABLE
mydb=> INSERT INTO beverages (price, serving_temp) VALUES (9.50, 'hot');
INSERT 0 1
mydb=> SELECT * FROM beverages;  -- a row has NULL name
   name    | price | serving_temp
-----------+-------+--------------
 Tea       |  9.99 | hot
 Lemonade  |  5.50 | cool
 Tea       |  9.50 | hot
           |  9.50 | hot
(4 rows)
```

3.6.5 5. Changing a Column's Default Value

This won't affect existing rows, only future insertions:

```
mydb=> ALTER TABLE beverages ALTER COLUMN name SET DEFAULT 'default_name';
ALTER TABLE
mydb=> INSERT INTO beverages (price, serving_temp) VALUES (9.50, 'hot');
INSERT 0 1
mydb=> SELECT * FROM beverages;
     name     | price | serving_temp
--------------+-------+--------------
 Tea          |  9.99 | hot
 Lemonade     |  5.50 | cool
 Tea          |  9.50 | hot
              |  9.50 | hot
 default_name |  9.50 | hot
(5 rows)
```

To remove a default value, use:

```
mydb=> ALTER TABLE beverages ALTER COLUMN name DROP DEFAULT;
ALTER TABLE
mydb=> INSERT INTO beverages (price, serving_temp) VALUES (9.50, 'hot');
INSERT 0 1
mydb=> SELECT * FROM beverages;
     name     | price | serving_temp
--------------+-------+--------------
 Tea          |  9.99 | hot
 Lemonade     |  5.50 | cool
 Tea          |  9.50 | hot
              |  9.50 | hot
 default_name |  9.50 | hot
              |  9.50 | hot
(6 rows)
```

`DROP DEFAULT` is equivalent to setting the default to `null`.

3.6.6 6. Changing a Column's Data Type

```
mydb=> ALTER TABLE beverages ALTER COLUMN price TYPE money;
ALTER TABLE
mydb=> SELECT * FROM beverages LIMIT 3;
   name    | price | serving_temp
-----------+-------+--------------
 Tea       | $9.99 | hot
 Lemonade  | $5.50 | cool
 Tea       | $9.50 | hot
(3 rows)
```

The `TYPE type_name` syntax only works if all existing entries in the column can be implicitly converted to the new type. For more complex conversions, add a `USING` clause to specify how to compute new values from the old:

```
mydb=> CREATE TYPE relative_temperature AS ENUM ('cold', 'cool', 'room', 'warm', 'hot');
CREATE TYPE
mydb=> ALTER TABLE beverages ALTER COLUMN serving_temp TYPE relative_temperature;
ERROR:  column "serving_temp" cannot be cast automatically to type relative_temperature
HINT:  You might need to specify "USING serving_temp::relative_temperature".
```

```
mydb=> ALTER TABLE beverages ALTER COLUMN serving_temp TYPE relative_temperature
mydb->   USING CAST (serving_temp AS relative_temperature);
ALTER TABLE
mydb=> INSERT INTO beverages VALUES ('Milk', 7.5, 'cool');
INSERT 0 1
mydb=> INSERT INTO beverages VALUES ('Water', 0.5, 'icy');
ERROR:  invalid input value for enum relative_temperature: "icy"
LINE 1: INSERT INTO beverages VALUES ('Water', 0.5, 'icy');
                                                    ^
```

PostgreSQL will attempt to convert default values and constraints as well, but it's often better to drop them first, and add suitably modified ones afterwards.

3.6.7 7. Renaming a Column

```
mydb=> ALTER TABLE beverages RENAME COLUMN price TO unit_price;
ALTER TABLE
mydb=> SELECT * FROM beverages LIMIT 3;
   name    | unit_price | serving_temp
-----------+------------+--------------
 Tea       |      $9.99 | hot
 Lemonade  |      $5.50 | cool
 Tea       |      $9.50 | hot
(3 rows)
```

3.6.8 8. Renaming a Table

```
mydb=> ALTER TABLE beverages RENAME TO drinks;
ALTER TABLE
mydb=> SELECT * FROM beverages;
ERROR:  relation "beverages" does not exist
LINE 1: SELECT * FROM beverages;
                      ^
```

```
mydb=> SELECT * FROM drinks LIMIT 3;
   name    | unit_price | serving_temp
-----------+------------+--------------
 Tea       |      $9.99 | hot
 Lemonade  |      $5.50 | cool
 Tea       |      $9.50 | hot
(3 rows)
```

3.7 Privileges

When an object is created, it is assigned an owner (usually the role that executed the creation statement).

You can assign ownership to another role if you are:

- a superuser or
- the current owner (or member of the owning role) **and** a member of the new owning role.

```
ALTER TABLE table_name OWNER TO new_owner;
```

Initially, only the owner or superusers can work with an object, unless privileges are granted. We use the `GRANT` command to assign privileges:

```
GRANT SELECT, UPDATE (details) ON staff_info TO hr_assistant;
```

Applicable privileges depend on the object's type. Writing `ALL` instead of a specific privilege grants all privileges relevant to the object type.

The special role `PUBLIC` can be used to grant a privilege to *every role* on the system. You can set up "group" roles to help manage privileges.

Use the `REVOKE` command to revoke previously granted privileges:

```
REVOKE SELECT, UPDATE (details) ON staff_info FROM hr_assistant;
```

Note: Ordinarily, only the object's owner (or a superuser) can grant or revoke privileges on an object. But if a privilege is granted **"with grant option"**, the recipient is allowed to grant it in turn to others.

If the grant option is revoked, all who received the privilege from that recipient (directly or through a chain of grants) will lose the privilege.

Owners are always treated as holding all grant options, and so can revoke and re-grant their own privileges.

- `SELECT`

  - Allows SELECT on all (or specific) columns of a table, view, materialized view or other table-like object.
  - Allows COPY TO.
  - Required to reference existing column values in UPDATE, DELETE or MERGE.
  - Allows currval function in sequences.
  - Allows large objects to be read.
- INSERT
  - Allows INSERT of new rows to all / specified columns.
  - Allows COPY FROM.
- UPDATE
  - Allows UPDATE on any / specified columns.
  - Typically requires the SELECT privilege to determine rows to update.
  - Required in SELECT ... FOR UPDATE and SELECT ... FOR SHARE.
  - Allows nextval and setval functions in sequences.
  - Allows writing and truncating large objects.
- DELETE
  - Allows DELETE on rows from a table, view, ...
  - Typically requires SELECT privilege - to reference table columns and determine what rows to delete.
- TRUNCATE
  - Allows TRUNCATE on a table.
- REFERENCES
  - Allows creation of a foreign key constraint.
- TRIGGER
  - Allows creation of a trigger on a table, view, ...
- CREATE
  - In databases:
    * allows creation of schemas and publications
    * allows installation of trusted extensions.
  - In schemas:
    * allows creation of new objects
    * allows renaming of existing objects you own.
  - In tablespaces:
    * allows creation of tables, indexes and temporary files
    * allows creation of databases with the tablespace as default.

Note: Revoking this privilege will not alter the existence or location of existing objects.

- CONNECT

  - Allows connection to the database.
  - Checked at connection startup.
- `TEMPORARY`
  - Allows creation of temporary tables.
- `EXECUTE`
  - Allows calling a function / procedure, including use of any operators implemented on top of the function.
- `USAGE`
  - For procedural languages:
    * allows use of the languages to create functions.
  - For schemas:
    * allows access to contained objects, assuming the objects' privilege requrements are met.

    > **Caution:** One can view object names even without this privilege e.g. by querying system catalogs.
    >
    > Revoking this permission is not a secure way to prevent object access since existing sessions might have statements that have previously performed the "look up".

  - For sequences:
    * allows use of `currval` and `nextval` functions.
  - For types and domains:
    * allows use in creation of tables, functions and other schema objects.
  - For foreign-data wrappers:
    * allows creation of new servers.
  - For foreign servers:
    * allows creation of foreign tables
    * allows creation, alteration or dropping of user mappings associated with the server.
- `SET`
  - Allows setting a server configuration parameter within the current session.
- `ALTER SYSTEM`
  - Allows setting a server configuration parameter using the `ALTER SYSTEM` command.

PostgreSQL grants default privileges when objects are created. These can be overridden using the `ALTER DEFAULT PRIVILEGES` command.

Default privileges always include all privileges for the owner, and can include some privileges for `PUBLIC` depending on the object type.

3.7.1 Demo

Listing 1: Creating users 'luther' and 'ethan' with the role 'agents'. *luther* can create databases (`createdb`).

```
$ createuser agents
$ createuser luther --role=agents --createdb --pwprompt
Enter password for new role:
Enter it again:
$ createuser ethan --role=agents --pwprompt
Enter password for new role:
Enter it again:
```

Listing 2: *luther* creates the 'top-secret' database and assigns it to *agents*. This allows all agents to access it (currently just *luther* and *ethan*).

```
$ createdb top-secret --username=luther --host=localhost --owner=agents
Password:
$ psql top-secret --username=luther --host=localhost
Password for user luther:
psql (15.3)
Type "help" for help.

top-secret=>
```

Listing 3: *luther* creates the 'agent_archive' table, becoming it's owner. *ethan* cannot access the *agent_archive* table until *luther* (or a superuser) grants him the necessary privileges.

```
top-secret=> CREATE TABLE agent_archive(
top-secret->   agent_id     serial PRIMARY KEY,
top-secret->   first_name   text,
top-secret->   last_name    text,
top-secret->   details      text
);
CREATE TABLE
top-secret=> INSERT INTO agent_archive (first_name, last_name, details)
top-secret->   VALUES ('Benjamin', 'Dunn', 'IT & logistics expert.');
INSERT 0 1
top-secret=> SELECT * FROM agent_archive;
 agent_id | first_name | last_name |        details
----------+------------+-----------+-----------------------
        1 | Benjamin   | Dunn      | IT & logistics expert.
(1 row)

top-secret=> \connect top-secret ethan
Password for user ethan:
You are now connected to database "top-secret" as user "ethan".
top-secret=> SELECT * FROM agent_archive;
ERROR:  permission denied for table agent_archive
```

Listing 4: *luther* grants the SELECT privilege to *ethan*. Now *ethan* can read from the *agent_archive* table. But *ethan* cannot add new values just yet.

```
top-secret=> \connect top-secret luther
Password for user luther:
You are now connected to database "top-secret" as user "luther".
top-secret=> GRANT SELECT ON agent_archive TO ethan;
GRANT
top-secret=> \connect top-secret ethan
Password for user ethan:
You are now connected to database "top-secret" as user "ethan".
top-secret=> SELECT * FROM agent_archive;
 agent_id | first_name | last_name |        details
----------+------------+-----------+-----------------------
        1 | Benjamin   | Dunn      | IT & logistics expert.
(1 row)

top-secret=> INSERT INTO agent_archive (first_name, last_name, details)
top-secret->   VALUES ('Ilsa', 'Faust', 'Ally. Mission Specialist.');
ERROR:  permission denied for table agent_archive
```

Listing 5: *luther* assigns ownership of the *agent_archive* table to the role *agents*. All agents inherit full rights to the table. *ethan* can now add info about his close ally.

```
top-secret=> \connect top-secret luther
Password for user luther:
You are now connected to database "top-secret" as user "luther".
top-secret=> ALTER TABLE agent_archive OWNER TO agents;
ALTER TABLE
top-secret=> \connect top-secret ethan
Password for user ethan:
You are now connected to database "top-secret" as user "ethan".
top-secret=> INSERT INTO agent_archive (first_name, last_name, details)
top-secret->   VALUES ('Ilsa', 'Faust', 'Ally. Mission Specialist.');
INSERT 0 1
top-secret=> SELECT * FROM agent_archive;
 agent_id | first_name | last_name |          details
----------+------------+-----------+---------------------------
        1 | Benjamin   | Dunn      | IT & logistics expert
        2 | Ilsa       | Faust     | Ally. Mission Specialist.
(2 rows)
```

3.7.2 Access Control List (ACL) Privilege Abbreviations

Privilege	Abbreviation	Applicable Object Types
SELECT	r (“read”)	LARGE OBJECT, SEQUENCE, TABLE (and table-like objects), table column
INSERT	a (“append”)	TABLE, table column
UPDATE	w (“write”)	LARGE OBJECT, SEQUENCE, TABLE, table column
DELETE	d	TABLE
TRUNCATE	D	TABLE
REFERENCES	x	TABLE, table column
TRIGGER	t	TABLE
CREATE	C	DATABASE, SCHEMA, TABLESPACE
CONNECT	c	DATABASE
TEMPORARY	T	DATABASE
EXECUTE	X	FUNCTION, PROCEDURE
USAGE	U	DOMAIN, FOREIGN DATA WRAPPER, FOREIGN SERVER, LANGUAGE, SCHEMA, SEQUENCE, TYPE
SET	s	PARAMETER
ALTER SYSTEM	A	PARAMETER

3.7.3 Summary of Access Privileges

Object Type	All Privileges	Default PUBLIC Privileges	psql Command
DATABASE	CTc	Tc	\l
DOMAIN	U	U	\dD+
FUNCTION or PROCEDURE	X	X	\df+
FOREIGN DATA WRAPPER	U	none	\dew+
FOREIGN SERVER	U	none	\des+
LANGUAGE	U	U	\dL+
LARGE OBJECT	rw	none	
SCHEMA	UC	none	\dn+
SEQUENCE	rwU	none	\dp
TABLE (and table-like objects)	arwdDxt	none	\dp
Table column	arwx	none	\dp
TABLESPACE	C	none	\db+
TYPE	U	U	\dT+

Assigned privileges are displayed as a list of `aclitem` entries. A * appears only when grant options have been explicitly granted.

3.8 Schemas

A database contains one or more schemas, which in turn contain tables, data types, functions, operators and other named objects. You can use the same object name in different schemas without conflict.

A client connection to the *postgres* server can only access a single database, specified in the connection request. But users can access objects in any of the schemas in the database, if granted privileges.

Use cases:

- To have multiple users in a database, without interference.
- To organise database objects into logical groups.
- To avoid name collisions i.e. from 3rd-party apps.

Schemas are analogous to directories in file-systems, but can't be nested.

3.8.1 Creating a Schema

Use the `CREATE SCHEMA` command with the desired schema name:

```
mydb=> CREATE SCHEMA services;
CREATE SCHEMA
```

The schema name can't start with `pg_` (system reserved). Omitting the schema name sets it same as the current user name.

You can create a schema owned by someone else e.g. to restrict user activities to well-defined namespaces:

```
CREATE SCHEMA some_schema AUTHORIZATION some_user;
```

To create or access objects in a specific schema, write a a *qualified name* e.g. `schema.object_name`, `database.schema.object_name`.

```
mydb=> CREATE TABLE services.deliveries (
mydb(>   transaction_id     text,
mydb(>   recepient_address  text,
mydb(>   date_dispatched    date,
mydb(>   completed_at       timestamp
mydb(> );
CREATE TABLE
mydb=> INSERT INTO services.deliveries (transaction_id, recepient_address, date_
↪dispatched, completed_at)
mydb->   VALUES ('12345', '678, abc way', '2023-08-10', '2023-08-10 15:06');
INSERT 0 1
```

3.8.2 Deleting a Schema

```
DROP SCHEMA some_schema;  -- if it's empty
DROP SCHEMA some_schema CASCADE;  -- drop all contained objects as well
```

3.8.3 The Schema Search Path

Database objects are often referred to by unqualified names for convenience. The system determines which object is meant by following a `search_path` - a list of schemas to look in. Then the first matching object is taken.

The first schema in `search_path` is the *current schema*. It is searched first.

```
mydb=> SHOW search_path;
   search_path
-----------------
 "$user", public
(1 row)
```

`$user` refers to the schema with the current user's name. If it doesn't exist (default), it's ignored.

An error is raised if no match is found, even if the object exists in some other schema.

```
mydb=> SELECT * FROM deliveries;  -- schema "services" not in search path
ERROR:  relation "deliveries" does not exist
LINE 1: SELECT * FROM deliveries;
                      ^
mydb=> SELECT * FROM services.deliveries;  -- schema-qualified name needed for now
 transaction_id | recepient_address | date_dispatched |    completed_at
----------------+-------------------+-----------------+---------------------
 12345          | 678, abc way      | 2023-08-10      | 2023-08-10 15:06:00
(1 row)
```

You can edit the schema search path with:

```
mydb=> SET search_path TO services,public;  -- search in services schema first
SET
mydb=> SELECT * FROM deliveries;
 transaction_id | recepient_address | date_dispatched |    completed_at
----------------+-------------------+-----------------+---------------------
 12345          | 678, abc way      | 2023-08-10      | 2023-08-10 15:06:00
(1 row)
```

The first schema that exists is the default location for creating new objects. This is why most objects are by default created in the `public` schema. There is nothing special about the public schema except that it exists by default. It can be dropped, too.

Important: In the SQL standard:

- There is no concept of a `public` schema. For maximum conformance to the standard, you should not use the `public` schema.
- The notion of objects in the same schema being owned by different users does not exist.

Caution: Due to the prevalence of unqualified names in queries and their use in PostgreSQL internals, adding a schema to *search_path* effectively trusts all users having CREATE privilege on that schema.

When you run an ordinary query, a malicious user able to create objects in a schema of your search path can take control and execute arbitrary SQL functions as though you executed them.

3.8.4 Privileges

Users can't access any objects in schemas they don't own (default), unless they're granted the `USAGE` privilege. The `CREATE` privilege is needed to create new objects.

For PostgreSQL 14 and below, all users have `CREATE` and `USAGE` privileges on the schema `public`. These can be revoked with:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
REVOKE USAGE ON SCHEMA public FROM PUBLIC;
```

3.8.5 The System Catalog Schema

Each database contains a **pg_catalog** schema, which contains the system tables and all the built-in data types, functions, and operators.

pg_catalog is always effectively part of the search path, to ensure built-in names are always findable. You can explicitly place *pg_catalog* at the end of your search path if you prefer to have user-defined names override built-in names.

3.8.6 Usage Patterns

A *secure schema usage pattern* prevents untrusted users from meddling with others' queries.

Options include:

- Constrain ordinary users to user-private schemas.
  - `REVOKE CREATE ON SCHEMA public FROM PUBLIC`.
  - Create a separate schema for each user, with the user's name so it's first in `search_path` (`$user`).
  - Secure unless untrusted user is the database owner or holds the `CREATEROLE` privilege.
- Remove `public` schema from the default search path.
  - Modify `postgresql.conf` or use `ALTER ROLE ALL SET search_path = $user`.
  - Now users must use qualified names to access/create objects in `public`.
  - Calls to functions in `public` are still unsafe.
  - Also secure unless untrusted user is the database owner or holds the `CREATEROLE` privilege.

In any case, to install shared applications (tables for everyone, additional functions from 3rd-parties, ...), put them in separate schemas, and grant appropriate privileges.

Tip: A quick hack to secure your queries would be to set `search_path` to `''`, or otherwise remove schemas other non-superusers can write to.

CHAPTER

FOUR

DATA MANIPULATION

4.1 Inserting Data

Data is inserted one row at a time. You can insert many rows in a single `INSERT` command.

Even if you supply values for a subset of the columns, a complete row must be created. The blank columns will be filled with their default values.

```
mydb=> CREATE TABLE farm_products (
mydb(>   name   text DEFAULT 'unnamed',
mydb(>   price  numeric(7,2),
mydb(>   units_in_stock  int
mydb(> );
CREATE TABLE
mydb=> INSERT INTO farm_products (name, price, units_in_stock) VALUES
mydb->   ('Carrots', 1.50, 64),
mydb->   ('Garlic', 2.00, 40);
INSERT 0 2
mydb=> SELECT * FROM farm_products;
  name   | price | units_in_stock
---------+-------+----------------
 Carrots |  1.50 |             64
 Garlic  |  2.00 |             40
(2 rows)
```

You can explicitly request default values for individual columns (using `DEFAULT`) or an entire row (using `DEFAULT VALUES`).

```
mydb=> INSERT INTO farm_products (name, units_in_stock) VALUES (DEFAULT, 0);
INSERT 0 1
mydb=> INSERT INTO farm_products DEFAULT VALUES;
INSERT 0 1
mydb=> SELECT * FROM farm_products;
  name   | price | units_in_stock
---------+-------+----------------
 Carrots |  1.50 |             64
 Garlic  |  2.00 |             40
 unnamed |       |              0
 unnamed |       |
(4 rows)
```

You can also insert the results of a query:

```
mydb=> SELECT * FROM products LIMIT 3;
   name   | items_in_stock | price
----------+----------------+-------
 Apples   |            100 | 25.00
 Bananas  |             32 | 10.00
 Cherries |             74 |  3.00
(3 rows)

mydb=> INSERT INTO farm_products (name, price, units_in_stock)
mydb->   SELECT name, price, items_in_stock FROM products LIMIT 3;
INSERT 0 3
mydb=> SELECT * FROM farm_products;
   name   | price | units_in_stock
----------+-------+----------------
 Carrots  |  1.50 |             64
 Garlic   |  2.00 |             40
 unnamed  |       |              0
 unnamed  |       |
 Apples   | 25.00 |            100
 Bananas  | 10.00 |             32
 Cherries |  3.00 |             74
(7 rows)
```

Tip: When inserting a lot of data at the same time, consider using the more efficient `COPY` command (or *psql*'s `\copy` command).

4.2 Updating Data

Use the `UPDATE` command.

```
mydb=> UPDATE farm_products SET price = price * 1.16;  -- add 16% VAT perhaps
UPDATE 7
mydb=> SELECT * FROM farm_products LIMIT 3;
  name   | price | units_in_stock
---------+-------+----------------
 Carrots |  1.74 |             64
 Garlic  |  2.32 |             40
 unnamed |       |              0
(3 rows)
```

You can add a `WHERE` clause to specify a condition that row(s) must meet in order to be updated:

```
mydb=> UPDATE farm_products SET name = 'unknown' WHERE name = 'unnamed';
UPDATE 2
mydb=> SELECT * FROM farm_products;
   name   | price | units_in_stock
----------+-------+----------------
 Carrots  |  1.74 |             64
 Garlic   |  2.32 |             40
```

(continues on next page)

(continued from previous page)

```
 Apples   | 29.00 |            100
 Bananas  | 11.60 |             32
 Cherries |  3.48 |             74
 unknown  |       |              0
 unknown  |       |
(7 rows)
```

You can update more than one column by listing more than one assignment in the SET clause:

```
mydb=> UPDATE farm_products SET price = 0, units_in_stock=0 WHERE name =
 'unknown';
UPDATE 2
mydb=> SELECT * FROM farm_products;
   name   | price | units_in_stock
----------+-------+----------------
 Carrots  |  1.74 |             64
 Garlic   |  2.32 |             40
 Apples   | 29.00 |            100
 Bananas  | 11.60 |             32
 Cherries |  3.48 |             74
 unknown  |  0.00 |              0
 unknown  |  0.00 |              0
(7 rows)
```

4.3 Deleting Data

Use the DELETE command.

You'll need to specify a condition that rows to be removed must match, or else all rows will be deleted.

```
mydb=> DELETE FROM farm_products WHERE name = 'unknown';
DELETE 2
mydb=> SELECT * FROM farm_products;
   name   | price | units_in_stock
----------+-------+----------------
 Carrots  |  1.74 |             64
 Garlic   |  2.32 |             40
 Apples   | 29.00 |            100
 Bananas  | 11.60 |             32
 Cherries |  3.48 |             74
(5 rows)

mydb=> DELETE FROM farm_products;  -- deletes all rows
DELETE 5
mydb=> SELECT * FROM farm_products;
 name | price | units_in_stock
------+-------+----------------
(0 rows)
```

4.4 Returning Data from Modified Rows

`INSERT`, `UPDATE` and `DELETE` commands have an optional `RETURNING` clause that avails data from modified rows while they're being manipulated (no need for an extra query to collect data).

Especially useful when it would be difficult to identify the modified rows reliably.

`RETURNING` clause contents are similar to a `SELECT` command's output list (column names, value expressions, ...).

- From an `INSERT`:
  - returns the row(s) as inserted
  - handy when relying on computed default values e.g. `SERIAL`:

```
mydb=> CREATE TABLE agents (
mydb(>   id    serial PRIMARY KEY,
mydb(>   first_name  text,
mydb(>   last_name   text,
mydb(>   status   text
mydb(> );
CREATE TABLE
mydb=> INSERT INTO agents (first_name, last_name, status) VALUES
mydb->   ('Ethan', 'Hunt', 'On field duty'),
mydb->   ('Luther', 'Stickell', 'On vacation')
mydb->   RETURNING id, last_name ||', ' || first_name AS name;
id |       name
----+------------------
  1 | Hunt, Ethan
  2 | Stickell, Luther
(2 rows)

INSERT 0 2
```

- From an `UPDATE`:
  - returns the new contents of the modified row(s).

```
mydb=> UPDATE agents SET status = 'On field duty' WHERE id = 2 RETURNING *;
id | first_name | last_name |    status
----+------------+-----------+---------------
  2 | Luther     | Stickell  | On field duty
(1 row)

UPDATE 1
```

- From a `DELETE`:
  - returns the contents of the deleted row(s).

```
mydb=> DELETE FROM agents RETURNING *;
id | first_name | last_name |    status
----+------------+-----------+---------------
  1 | Ethan      | Hunt      | On field duty
  2 | Luther     | Stickell  | On field duty
(2 rows)
```

(continues on next page)

(continued from previous page)

```
DELETE 2
```

Note: If there are *triggers* on the target table, the data available to `RETURNING` is the row as modified by the triggers. Thus, inspecting columns computed by triggers is another common use-case for `RETURNING`.

CHAPTER

FIVE

QUERIES

5.1 Table Expressions

A table expression computes a table. It contains a `FROM` clause that is optionally followed by `WHERE`, `GROUP BY`, and `HAVING` clauses.

5.1.1 1. The FROM Clause

Derives a table from one or more other tables given in a table reference list:

```
FROM table_reference [, table_reference [, ...]]
```

A table reference can be a (schema-qualified) table name or a derived table (e.g. a subquery, a `JOIN` construct).

If more than one table reference is listed, the tables are **cross-joined** - the Cartesian product of their rows is formed. The result is a virtual table.

For tables which are parents, the table reference produces rows of all descendants, unless the keyword `ONLY` precedes the table name. Either way, only columns in the parent are produces, and those added in subtables are ignored.

1.1 Joined Tables

Derived from two other (real or derived) tables. The general syntax is:

```
T1 join_type T2 [ join_condition ]
```

Joins can be chained or nested. `()` can be used to control join order, or else they're evaluated left-to-right.

1.1.1 Cross Join

Rows produced are a Cartesian product (every possible combination) of T1 and T2.

```
T1 CROSS JOIN T2 | T1, T2 | T1 INNER JOIN T2 ON TRUE
```

If T1 had N rows and T2 had M rows, N * M rows are produced.

```
CREATE TABLE T1 (num  int, name  char(1));
INSERT INTO T1 VALUES (1, 'a'), (2, 'b'), (3, 'c');
CREATE TABLE T2 (num  int, letters  char(3));
INSERT INTO T2 VALUES (1, 'xxx'), (3, 'yyy'), (5, 'zzz');
```

```
mydb=> SELECT * FROM T1, T2;
 num | name | num | letters
-----+------+-----+---------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)
```

Note: Joins bind more tightly than commas. `FROM T1 CROSS JOIN T2 INNER JOIN T3 ON condition` is not the same as `FROM T1, T2 INNER JOIN T3 ON condition`, since condition can reference T1 in the first case but not the second.

1.1.2 Qualified Joins

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( join column list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

`INNER`(default) and `OUTER` are optional. `LEFT`, `RIGHT` and `FULL` imply an outer join.

The *join_condition* is specified in `ON`, `USING` or `NATURAL`. It determines which rows from the 2 source tables are considered to "match".

- `INNER JOIN`:
  - Only rows that "match" in both tables are included.

```
mydb=> SELECT * FROM T1 INNER JOIN T2 ON T1.num = T2.num;
 num | name | num | letters
-----+------+-----+---------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)
```

- `LEFT OUTER JOIN`:
  - An inner join is performed.
  - Rows in T1 with no "match" in T2 are added, with null values in columns of T2.

```
mydb=> SELECT * FROM T1 LEFT JOIN T2 USING (num);
 num | name | letters
-----+------+---------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)
```

- `RIGHT OUTER JOIN`:

  - An inner join is performed.
  - Rows in T2 with no "match" in T1 are added, with null values in columns of T1.

```
mydb=> SELECT * FROM T1 RIGHT JOIN T2 USING (num);
 num | name | letters
-----+------+---------
   1 | a    | xxx
   3 | c    | yyy
   5 |      | zzz
(3 rows)
```

- `FULL OUTER JOIN`:
  - An inner join is performed.
  - Rows in T1 with no "match" in T2 are added, with null values in columns of T2.
  - Rows in T2 with no "match" in T1 are added, with null values in columns of T1.

```
mydb=> SELECT * FROM T1 FULl JOIN T2 USING (num);
 num | name | letters
-----+------+---------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
   5 |      | zzz
(4 rows)
```

The `ON` clause takes a boolean value expression, same as in WHERE. A pair of rows from T1 and T2 match if the expression evaluates to true.

The `USING` clause is used when both sides of the join use the *same name(s) for the joining column(s)*. It takes a comma-separated list of the shared column names. Joining T1 and T2 with `USING (a, b)` produces the join condition `ON T1.a = T2.a AND T1.b = T2.b`.

Importantly, only one of the shared columns is returned in `JOIN USING`, whereas `JOIN ON` still includes them.

`NATURAL` is a shorthand form of `USING` that forms a USING list consisting of *all column names that appear in both tables*. If there are no common names, `NATURAL JOIN` behaves like `JOIN ... ON TRUE`, resulting in a cross join.

> **Note:** A restriction placed on the `ON` clause is processed before the join, while one placed in a `WHERE` clause is processed after the join. This matters a lot in outer joins:

```
mydb=> SELECT * FROM T1 LEFT JOIN T2 ON T1.num = T2.num AND T2.letters = 'xxx';
 num | name | num | letters
-----+------+-----+---------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)

mydb=> SELECT * FROM T1 LEFT JOIN T2 ON T1.num = T2.num WHERE T2.letters = 'xxx';
 num | name | num | letters
-----+------+-----+---------
   1 | a    |   1 | xxx
(1 row)
```

1.2 Table & Column Aliases

Table aliases are temporary names given to tables, mainly for notational convenience. You may not refer to the original name afterwards.

```
FROM table_reference [AS] alias
FROM table_reference [AS] alias ( column1 [, column2 [, ...]
```

`AS` is optional. `alias` can be any identifier.

The second form gives temporary names to the table as well as its columns. If fewer column aliases are specified, the remaining columns are not renamed.

You **must** use an alias when joining a table to itself, or if the table reference is a subquery:

```
SELECT
    *
FROM
    people AS parent
JOIN people AS child ON
    parent.id = child.parent_id;
```

1.3 Subqueries

Subqueries specifying a derived table must be enclosed in `()` and assigned a table alias.

```
FROM (SELECT ... FROM some_table) AS alias_name
```

A subquery can also be a `VALUES` list:

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
    AS names(first, last)
```

1.4 Table Functions

These are functions that produce a set of rows of either base (scalar) data types or composite data types (table rows).

Columns from tabke functions can be used in `SELECT`, `JOIN` or `WHERE` clauses just like tables, views or subqueries.

Table functions may be combined using `ROWS FROM`, returning parallell columns; number of rows is that of the largest function result, with smaller results padded with null.

```
function_call [WITH ORDINALITY] [[AS] table_alias [(column_alias [, ... ])]]
ROWS FROM( function_call [, ... ] ) [WITH ORDINALITY] [[AS] table_alias [(column_alias [,
↪ ... ])]]
```

Using `WITH ORDINALITY` adds a `bigint` column numbering the columns of the function result set, starting from 1, named 'ordinality' (default).

The special table function `UNNEST` acts on array parameters, returning columns as if `UNNEST` had been called on each parameter separately and combined using `ROWS FROM`:

```
UNNEST( array_expression [, ... ] ) [WITH ORDINALITY] [[AS] table_alias [(column_alias [,
↪ ... ])]]
```

If no table_alias is set, the function name is used as the table name. In `ROWS FROM ()`, the function's first name is used.

If column aliases are not supplied, then for a function returning a base data type, the column name is also the same as the function name. For a function returning a composite type, the result columns get the names of the individual attributes of the type.

1.5 Lateral Subqueries

Preceding subqueries in `FROM` with the key word `LATERAL` allows them to reference columns provided by preceding items. Without `LATERAL`, each subquery is evaluated independently.

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id = foo.bar_id) ss;
```

`LATERAL` is optional in table functions since they can reference preceding items anyways.

A LATERAL item can appear at top level in the FROM list, or within a JOIN tree.

`FROM` items containing `LATERAL` cross-references are processed as:

- For each row of the `FROM` item providing the cross-refereced column(s), or set of rows of mutliple `FROM` items providing the columns, the `LATERAL` item is evaluated using that row or row set's values of the columns.
- The resulting row(s) are joined as usual with the rows they were computed from.
- Repeat for every row or set of rows from the column source table(s).

`LATERAL` is primarily useful when the cross-referenced column is necessary for computing the row(s) to be joined.

It is often particularly handy to `LEFT JOIN` to a `LATERAL` subquery, so that source rows will appear in the result even if the `LATERAL` subquery produces no rows for them.

5.1.2 2. The WHERE Clause

Filters rows of the derived virtual table from `FROM`.

```
WHERE search_condition
```

where *search_condition* is any value expression that returns a boolean value. Only rows evaluating to true are kept (false, null are discarded).

> **Note:** The join condition of an inner join can be written in the `WHERE` clause or in the `JOIN` clause:

```
FROM a, b WHERE a.id = b.id AND b.val > 5
-- is equivalent to
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

5.1.3 3. GROUP BY & HAVING Clauses

`GROUP BY` groups rows in a table with similar values in the listed columns, to eliminate redundancy in the output and/or compute aggregates.

In general, if a table is grouped, columns not listed in `GROUP BY` cannot be referenced except in aggregate expressions.

In strict SQL, `GROUP BY` can only group by columns of the source table. But *postgres* columns in the select list and value expressions.

`HAVING` can be used to include only groups of interest. Expressions in a HAVING clause can refer to grouped expressions ungrouped expressions involving an aggregate function.

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

If a query contains aggregate function calls, but no `GROUP BY` clause, grouping still occurs resulting in a single group row. The same is true if it contains a `HAVING` clause, even without any aggregate function calls or GROUP BY clause.

5.1.4 4. GROUPING SETS, CUBE & ROLLUP

Grouping sets group rows just like `GROUP BY` clauses.

```
CREATE TABLE items (
    brand       varchar(20),
    size        varchar(3),
    sales       decimal(7, 2)
);
INSERT INTO items
    VALUES ('Foo', 'L', 10), ('Foo', 'M', 20), ('Bar', 'M', 15), ('Bar', 'L',  5);
```

```
mydb=> SELECT * FROM items;
 brand | size | sales
-------+------+-------
 Foo   | L    | 10.00
 Foo   | M    | 20.00
 Bar   | M    | 15.00
 Bar   | L    |  5.00
(4 rows)

mydb=> SELECT brand, size, sum(sales) FROM items GROUP BY GROUPING SETS ((brand), (size),
↪ ());
 brand | size |  sum
-------+------+-------
       |      | 50.00
 Foo   |      | 30.00
 Bar   |      | 20.00
       | L    | 15.00
       | M    | 35.00
(5 rows)
```

Each sublist of `GROUPING SETS` may specify zero or more columns / expressions, and is interpreted as if directly in a `GROUP BY` clause.

An empty grouping set `()` means that all rows are aggregated down to a single group.

References to the grouping columns / expressions are replaced by null values in result rows for grouping sets in which those columns do not appear.

```
ROLLUP ( e1, e2, e3, ... )
-- is equivalent to
GROUPING SETS (
    ( e1, e2, e3, ... ),
    ...
    ( e1, e2 ),
    ( e1 ),
    ( )
)
```

ROLLUP is commonly used for analysis over heirachical data e.g. total salary by department, division and company-wide total.

```
CUBE ( a, b, c )
-- is equivalent to
GROUPING SETS (
    ( a, b, c ),
    ( a, b    ),
    ( a,    c ),
    ( a       ),
    (    b, c ),
    (    b    ),
    (       c ),
    (         )
) -- power set(all possible subsets)
```

Sublist elements in CUBE and ROLLUP clauses are treated as single units:

```
CUBE ( (a, b), (c, d) )
-- is equivalent to
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b       ),
    (       c, d ),
    (            )
)

ROLLUP ( a, (b, c), d )
-- is equivalent to
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b, c    ),
    ( a          ),
    (            )
)
```

CUBE and ROLLUP can either be used directly in GROUP BY, or nested inside a GROUPING SETS clause.

Nesting a GROUPING SET clause inside another treats all elements of the inner clause as if directly written in the outer clause.

If multiple grouping items are specified in a single GROUP BY, the final list of grouping sets is the cross product of the individual items:

```
GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))
-- is equivalent to
GROUP BY GROUPING SETS (
    (a, b, c, d), (a, b, c, e),
    (a, b, d),    (a, b, e),
    (a, c, d),    (a, c, e),
    (a, d),       (a, e)
)
```

The final set of grouping sets might contain duplicates, which can be removed using the `DISTINCT` clause directly on the `GROUP BY`:

```
GROUP BY ROLLUP (a, b), ROLLUP (a, c)
-- is equivalent to
GROUP BY GROUPING SETS (
    (a, b, c),
    (a, b),
    (a, b),
    (a, c),
    (a),
    (a),
    (a, c),
    (a),
    ()
)

GROUP BY DISTINCT ROLLUP (a, b), ROLLUP (a, c)
-- is equivalent to
GROUP BY GROUPING SETS (
    (a, b, c),
    (a, b),
    (a, c),
    (a),
    ()
)
```

5.1.5 5. Window Function Processing

Window functions are evaluated after grouping, aggregation and `HAVING` filtering are performed. They won't see the original rows from `FROM/WHERE`.

Multiple window functions having syntactically equivalent `PARTITION BY` and `ORDER BY` clauses are guaranteed to be evaluated in a single pass over the data.

Currently, window functions always require presorted data, and so query output will be ordered according to one or another of the window functions' `PARTITION BY/ORDER BY` clauses. Use an explicit top-level `ORDER BY` if you wish to guarantee a particular order.

5.2 UNION, INTERSECT & EXCEPT

The results of 2 queries can be combined using the set operations *union*, *intersection*, and *difference*:

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

The queries must be **"union compatible"**:

- both return the same number of columns
- corresponding columns have compatible data types.

```
mydb=> CREATE TABLE fruits (name text, price money);
CREATE TABLE
mydb=> INSERT INTO fruits VALUES ('Apples', 25), ('Tomatoes', 10), ('Pears', 16);
INSERT 0 3
mydb=> CREATE TABLE vegetables (name text, price money);
CREATE TABLE
mydb=> INSERT INTO vegetables VALUES ('Spinach', 5), ('Carrots', 4), ('Tomatoes', 10);
INSERT 0 3
```

5.2.1 UNION

Appends the result of query2 to the result of query 1 (order of returned rows still not guaranteed). *Eliminates duplicate rows* (like `DISTINCT`), unless `UNION ALL` is used.

```
mydb=> SELECT name, price FROM fruits
mydb->   UNION SELECT name, price FROM vegetables;
   name    | price
-----------+--------
 Apples    | $25.00
 Carrots   |  $4.00
 Pears     | $16.00
 Spinach   |  $5.00
 Tomatoes  | $10.00
(5 rows)
```

```
mydb=> SELECT name, price FROM fruits
mydb->   UNION ALL SELECT name, price FROM vegetables;
   name    | price
-----------+--------
 Apples    | $25.00
 Tomatoes  | $10.00
 Pears     | $16.00
 Spinach   |  $5.00
 Carrots   |  $4.00
 Tomatoes  | $10.00
(6 rows)
```

5.2.2 INTERSECT

Returns rows *present in both* query1 and query2 results. *Eliminates duplicate rows*, unless `INTERSECT ALL` is used.

```
mydb=> SELECT name, price FROM fruits
mydb->   INTERSECT SELECT name, price FROM vegetables;
   name    | price
-----------+--------
 Tomatoes | $10.00
(1 row)
```

5.2.3 EXCEPT

Returns rows present in the result of query1 but not that of query2 (aka *difference*). *Eliminates duplicates*, unless `EXCEPT ALL` is used.

```
mydb=> SELECT name, price FROM fruits
mydb->   EXCEPT SELECT name, price FROM vegetables;
  name  | price
--------+--------
 Apples | $25.00
 Pears  | $16.00
(2 rows)
```

Note: You might need to surround individual queries with parentheses e.g. if any of the queries has a LIMIT clause.

```
mydb=> SELECT * FROM fruits ORDER BY price LIMIT 2
mydb->   UNION SELECT * FROM vegetables;
ERROR:  syntax error at or near "UNION"
LINE 2:   UNION SELECT * FROM vegetables;
          ^
```

```
mydb=> (SELECT * FROM fruits ORDER BY price LIMIT 2)
mydb->   UNION SELECT * FROM vegetables;
   name    | price
-----------+--------
 Carrots  |  $4.00
 Pears    | $16.00
 Spinach  |  $5.00
 Tomatoes | $10.00
(4 rows)
```

Set operations can be combined. You can use `()` to control evaluation order:

```
query1 UNION query2 EXCEPT query3
-- is equivalent to
(query1 UNION query2) EXCEPT query3
```

Important: Without parentheses, `UNION` and `EXCEPT` associate left-to-right, but `INTERSECT` binds more tightly than these 2:

```
query1 UNION query2 INTERSECT query3
-- is equivalent to
query1 UNION (query2 INTERSECT query3)
```

5.3 ORDER BY

Sorts rows in the output table.

```
SELECT select_list
    FROM table_expression
    ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
             [, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

The *sort_expression(s)* can be any expression valid in a select list e.g. `col1 + col2`. When more than one expression is specified, the later values are used to sort rows that are equal according to the earlier values.

Without sorting, rows are returned in an unspecified order (no guarantee):

```
mydb=> SELECT * FROM drinks;
   name    | unit_price | serving_temp
-----------+------------+--------------
 Tea       |      $9.99 | hot
 Lemonade  |      $5.50 | cool
 Milk      |      $7.50 | cool
 Coffee    |            | hot
 Tea       |      $9.50 | warm
(5 rows)
```

`ASC` and `DESC` keywords set the sort direction to ascending and descending respectively:

```
mydb=> SELECT * FROM drinks ORDER BY name, serving_temp;  -- ASC is default
   name    | unit_price | serving_temp
-----------+------------+--------------
 Coffee    |            | hot
 Lemonade  |      $5.50 | cool
 Milk      |      $7.50 | cool
 Tea       |      $9.50 | warm
 Tea       |      $9.99 | hot
(5 rows)
```

```
mydb=> SELECT * FROM drinks ORDER BY name DESC, serving_temp DESC;
   name    | unit_price | serving_temp
-----------+------------+--------------
 Tea       |      $9.99 | hot
 Tea       |      $9.50 | warm
 Milk      |      $7.50 | cool
 Lemonade  |      $5.50 | cool
 Coffee    |            | hot
(5 rows)
```

Note: Ordering options are *considered independently* for each sort colums: `ORDER BY col1, col2 DESC` means `ORDER BY col1 ASC, col2 DESC`.

`NULLS FIRST` and `NULLS LAST` options can be used to determine whether nulls appear before or after non-null values. By default, null values sort as if larger than any non-null value. So, NULLS FIRST is default in DESC; NULLS LAST in ASC.

```
mydb=> SELECT * FROM drinks ORDER BY unit_price DESC;
   name    | unit_price | serving_temp
-----------+------------+--------------
 Coffee    |            | hot
 Tea       |      $9.99 | hot
 Tea       |      $9.50 | warm
 Milk      |      $7.50 | cool
 Lemonade  |      $5.50 | cool
(5 rows)

mydb=> SELECT * FROM drinks ORDER BY unit_price DESC NULLS LAST;
   name    | unit_price | serving_temp
-----------+------------+--------------
 Tea       |      $9.99 | hot
 Tea       |      $9.50 | warm
 Milk      |      $7.50 | cool
 Lemonade  |      $5.50 | cool
 Coffee    |            | hot
(5 rows)
```

A sort expression can also be the *alias* or *number* of an output column:

```
mydb=> SELECT initcap(serving_temp::text) || ' ' || name AS drink, unit_price
mydb->   FROM drinks
mydb->   ORDER BY drink, 2;
     drink     | unit_price
---------------+------------
 Cool Lemonade |      $5.50
 Cool Milk     |      $7.50
 Hot Coffee    |
 Hot Tea       |      $9.99
 Warm Tea      |      $9.50
(5 rows)
```

5.4 LIMIT & OFFSET

```
SELECT select_list
    FROM table_expression
    [ ORDER BY ... ]
    [ LIMIT { number | ALL } ] [ OFFSET number ]
```

`LIMIT` specifies that no more than *number* rows should be returned (can be less). `LIMIT ALL` and `LIMIT NULL` are equivalent to omitting the LIMIT clause.

Important: When using `LIMIT`, it is essential to use an `ORDER BY` clause, or else you'll get unpredictable subsets of rows.

```
mydb=> SELECT * FROM drinks ORDER BY unit_price LIMIT 3;
   name    | unit_price | serving_temp
-----------+------------+--------------
 Lemonade  |      $5.50 | cool
 Milk      |      $7.50 | cool
 Tea       |      $9.50 | warm
(3 rows)
```

```
mydb=> SELECT * FROM drinks ORDER BY unit_price LIMIT 10;  -- drinks has only 5 rows
   name    | unit_price | serving_temp
-----------+------------+--------------
 Lemonade  |      $5.50 | cool
 Milk      |      $7.50 | cool
 Tea       |      $9.50 | warm
 Tea       |      $9.99 | hot
 Coffee    |            | hot
(5 rows)
```

`OFFSET` specifies how many rows to skip before returning rows. `OFFSET 0` and `OFFSET NULL` are equivalent to omitting the OFFSET clause.

```
mydb=> SELECT * FROM drinks ORDER BY unit_price LIMIT 2 OFFSET 2;
 name | unit_price | serving_temp
------+------------+--------------
 Tea  |      $9.50 | warm
 Tea  |      $9.99 | hot
(2 rows)
```

Note: The *query optimizer* takes `LIMIT` into account when generating query plans.

A large `OFFSET` *may be inefficient* since the rows skipped still have to be computed inside the server.

CHAPTER

SIX

DATA TYPES

6.1 Numeric Types

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
decimal	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	variable-precision, inexact	6 decimal digits precision
double precision	8 bytes	variable-precision, inexact	15 decimal digits precision
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

6.1.1 1. Integer Types

`smallint`, `integer` and `bigint` store whole numbers (without fractional parts).

Attempting to store a value outside the allowed range throws an error.

`integer` (`int`) offers the best range-storage-performance balance.

6.1.2 2. Arbitrary Precision Numbers

`numeric` and `decimal` are equivalent, and are both part of the SQL standard. They are especially recommended for storing quantities where exactness is required e.g money.

Calculations on `numeric` values yields exact results where possible, but are relatively much slower than in integer or floating-point types.

- **precision**: the total count of *significant digits* to both sides of the decimal point. Must be positive.
- **scale**: the count of *decimal digits* in the fractional part. Positive or zero.

```
NUMERIC(precision, scale)
NUMERIC(precision)    -- zero scale
NUMERIC    -- unconstrained
```

> **NOTE:** The maximum precision that can be explicitly specified in a `NUMERIC` type declaration is 1000.
>
> An unconstrained `NUMERIC` is subject to the implementation limits in the table above.

The SQL standard requires a default scale of 0 (coercion to integer precision), so always specify precision and scale to ensure portability.

Values with a larger scale than that set will be rounded to the set scale. Then, if the new precison exceeds that declared, an error is raised.

> **NOTE:** `Numeric` values are stored *without extra leading or trailing zeroes*. The declared precison and scale are maximums, not fixed allocations (akin to `varchar`).
>
> The actual storage requirement is 2 bytes per 4 decimal digits, plus a 3 to 8 byte overhead.

The `numeric` type also includes the special values `'Infinity'` (`'inf'`), `'-Infinity'` (`'-inf'`) and `'NaN'`, case insensitive.

```
inf + x = inf
inf + inf = inf
inf - inf = NaN
x / inf = 0
```

`NaN` is used to represent undefined calculation results. Operations with a `NaN` input yield another `NaN`, with some exceptions e.g. `NaN ^ 0`.

> **NOTE:** In most implementations, `NaN` is considered not equal to any other numeric value (including `NaN`).
>
> In order to allow numeric values to be *sorted* and used in *tree-based indexes*, *PostgreSQL* treats `NaN` values as *equal, and greater than all non-`NaN` values*.

When rounding values, the `numeric` type rounds ties *away from zero*, while float types round ties to the nearest even number:

```
mydb=> SELECT x,
mydb->        round(x::numeric) AS numeric_round,
mydb->        round(x::double precision) AS double_round
mydb->   FROM generate_series(-3.5, 3.5, 1) as x;
  x   | numeric_round | double_round
------+---------------+--------------
 -3.5 |            -4 |           -4
 -2.5 |            -3 |           -2
 -1.5 |            -2 |           -2
 -0.5 |            -1 |           -0
```

(continues on next page)

(continued from previous page)

```
  0.5 |             1 |             0
  1.5 |             2 |             2
  2.5 |             3 |             2
  3.5 |             4 |             4
(8 rows)
```

6.1.3 3. Floating-Point Types

`real` and `double precision` are *inexact, variable-precision* numeric types - implementations of IEEE Standard 754 for *Binary Floating-Point Arithmetic* (*single* and *double precision*, respectively).

Some values can't be converted exactly to the internal format, and are stored as approximations, such that storing and retrieving a value might show slight discrepancies.

`real` has a range of around `1E-37 to E+37`, with a precision of at least 6 decimal digits.

`double precision` has a range of `1E-307 to 1E+308`, with a precision of at least 15 digits.

Values that are too large/small raise an error.

Input values with excess precison might be rounded.

Numbers too close to zero that are not representable as distinct from zero will cause an *underflow error*.

By default, floating point values are output in text form in their shortest precise decimal representation:

```
mydb=> SELECT 4.213327242424::real;
  float4
-----------
 4.2133274
(1 row)
```

The `extra_floats_digits` parameter can be used to select the rounded decimal output. Setting zero restores the default. Negative values reduce significant decimals, and positive values select the shortest-precise format.

Floating-point types also include special values `'Infinity'` (`'inf'`), `'-Infinity'` (`'-inf'`) and `'NaN'`.

PostgreSQL also supports the SQL-standard notations `float` and `float(p)` for specifying *inexact numeric types*, where `p` specifies the *minimum acceptable precision* in binary digits.

`float(1)` to `float(24)` select the `real` type. `float(25)` to `float(53)` select `double precision`. Values of `p` outside the allowed range draw an error.

`float` with no precision specified is taken to mean `double precision`.

6.1.4 4. Serial Types

`smallserial`, `serial`, and `bigserial` are not true types, but notational convenience for creating *unique identifier columns* (similar to `AUTO_INCREMENT`).

The tables from the queries below are equivalent:

```
CREATE TABLE table1 (
    col1 SERIAL
);
```

(continues on next page)

(continued from previous page)

```
CREATE SEQUENCE tablename_colname_seq AS integer;
CREATE TABLE table2 (
    col1 integer NOT NULL DEFAULT nextval('tablename_colname_seq')
);
ALTER SEQUENCE tablename_colname_seq OWNED BY table2.col1;
```

i.e:

- create a sequence
- create an integer column whose default values are assigned from a sequence generator
- Add constraints e.g `PRIMARY KEY` to ensure values are unique and non-null.
- mark the sequence as owned by the column, so that it will be dropped if the column or table is dropped.

  NOTE: Because `smallserial`, `serial` and `bigserial` are implemented using *sequences*, there may be gaps in the sequence of values which appears in the column, even if no rows are ever deleted.

  A value allocated from the sequence is still "used up" even if a row containing that value is never successfully inserted into the table column e.g in rolled back transactions.

To insert a value into a `serial` column, either exclude it from the list of columns or use the `DEFAULT` keyword.

`serial` and `serial4` are equivalent: both create `integer` columns.

`bigserial` and `serial8` create `bigint` columns.

`smallserial` and `serial2` create `smallint` columns.

6.2 Monetary Types

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

`money` stores currency amounts with a *fixed fractional precision*.

The `lc_monetary` setting determines the locale to use for formatting monetary amounts.

Input for `money` is accepted in a variety of numeric formats, including typical currency formatting such as '$25'. Output depends on locale.

```
mydb=> SET lc_monetary='sw_KE.utf8';
SET
mydb=> SELECT '25000'::money;
    money
--------------
 Ksh25,000.00
(1 row)
```

> **NOTE:** Since the output is *locale-sensitive*, it might not work to load `money` data into a database that has a *different setting* of `lc_monetary`.
>
> Before restoring a dump into a new database, make sure `lc_monetary` has the *same or equivalent value* as in the database that was dumped.

`numeric`, `int` and `bigint` types can be directly cast to `money`, but `real` and `double precision` have to be cast to `numeric` first.

A `money` value can be cast to `numeric` without loss of precision. Conversion to other types involves intermediate conversion to `numeric`, and could potentially lose precision.

Division of a `money` value with an `int` involves *truncation* of the fractional part towards zero. To avoid losing precision, cast the value to `numeric` before dividing and back to `money` afterwards.

When a `money` value is divided by another `money` value, the result is `double pecision` and not `money`. The currency units cancel each other.

6.3 Character Types

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

Name	Storage Size	Description
“char”	1 byte	single-byte internal type
name	64 bytes	internal type for object names

SQL defines 2 primary character types - `character varying(n)` (`varchar(n)`) and `character(n)` (`char(n)`).

Both store strings up to `n` characters in length (n must be positive).

Strings longer than expected raise an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length.

Explicitly casting a value to `char(n)` or `varchar(n)` silently truncates over-length values to `n` characters.

The database character set is selected when the database is created. The character with code zero can’t be stored.

Short strings (up to 126 bytes) have an overhead of 1 byte. Long strings have an overhead of 4 bytes.

Long strings are *automatically compressed*, to save disk space.

Very long values are stored in *background tables* to ensure rapid access to shorter values.

The longest possible character string that can be stored is 1 GB.

6.3.1 1. Variable-Length Types

1.1 Character Varying(n)

Strings shorter than declared are *stored as they are*.

Trailing spaces are *semantically significant*, as in `text` values and *pattern matching*.

Without `n`, accepts strings of any size (*postgres* extension).

```
mydb=> CREATE TABLE test2 (b varchar(5));
CREATE TABLE
mydb=> INSERT INTO test2 VALUES ('ok'), ('good       ');
INSERT 0 2
mydb=> INSERT INTO test2 VALUES ('too long');
ERROR:  value too long for type character varying(5)
mydb=> INSERT INTO test2 VALUES ('too long'::varchar(5));
INSERT 0 1
mydb=> SELECT b, char_length(b) FROM test2;
   b   | char_length
-------+-------------
 ok    |           2
 good  |           5
 too l |           5
(3 rows)
```

1.2 Text

The `text` type stores strings of *any length*.

It is not standard SQL, but has been implemented by several other DBMS as well.

6.3.2 2. Fixed-Length Types

2.1 Character(n)

Strings shorter than declared are *space-padded*. They are stored and displayed this way.

Trailing spaces are treated as *semantically insignificant*, and disregarded in `char(n)` - `char(n)` comparisons.

Trailing spaces are removed when converting to other string types.

Without `n`, is equivalent to `char(1)`.

```
mydb=> CREATE TABLE test1 (a character(4));
CREATE TABLE
mydb=> INSERT INTO test1 VALUES ('ok');
INSERT 0 1
mydb=> INSERT INTO test1 VALUES ('good  ');
INSERT 0 1
mydb=> INSERT INTO test1 VALUES ('too long  ');
ERROR:  value too long for type character(4)
mydb=> SELECT a, char_length(a) FROM test1;
  a   | char_length
------+-------------
 ok   |           2
 good |           4
(2 rows)
```

2.2 Name

Not for general use. Exists only for the storage of identifiers in internal system catalogs.

2.3 “Char”

Used internally as a simplistic enumeration type in system catalogs.

CHAPTER

SEVEN

FUNCTIONS & OPERATORS

7.1 Logical Operators

SQL uses a 3-valued logic system with `TRUE`, `FALSE` and `NULL` (unknown):

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	NULL

The operators `AND` and `OR` are commutative. However, it is not guaranteed that the left operand is evaluated before the right.

7.2 Comparison Functions & Operators

7.2.1 1. Comparison Operators

Operator	Description
datatype < datatype → boolean	Less than
datatype > datatype → boolean	Greater than
datatype <= datatype → boolean	Less than or equal to
datatype >= datatype → boolean	Greater than or equal to
datatype = datatype → boolean	Equal
datatype <> datatype → boolean	Not equal
datatype != datatype → boolean	Not equal

NOTE: <> is the SQL notation for *not equal*. != is an alias, converted to <> at a very early stage of parsing.

Comparison operators are available for all built-in data types that have a *natural ordering* (numeric, string, date/time, …).

Arrays, composite types and ranges can be compared if their component data types are comparable.

It's possible to compare values of *related data types* e.g. `interger < bigint`, by either:

- *cross-type* comparison operators, if available
- coercing the less general type to the more general during parsing

All comparison operators are *binary operators* that return `boolean` values, so expressions like `1 < 2 < 3` are *not valid*. Use `BETWEEN` to perform range tests.

7.2.2 2. Comparison Predicates

Predicate	Description	Example(s)
datatype BETWEEN datatype AND datatype → boolean	Between (inclusive of the range endpoints).	2 BETWEEN 1 AND 3 → t 2 BETWEEN 3 AND 1 → f
datatype NOT BETWEEN datatype AND datatype → boolean	Not between (the negation of BETWEEN).	2 NOT BETWEEN 1 AND 3 → f
datatype BETWEEN SYMMETRIC datatype AND datatype → boolean	Between, after sorting the two endpoint values.	2 BETWEEN SYMMETRIC 3 AND 1 → t
datatype NOT BETWEEN SYMMETRIC datatype AND datatype → boolean	Not between, after sorting the two endpoint values.	2 NOT BETWEEN SYMMETRIC 3 AND 1 → f
datatype IS DISTINCT FROM datatype → boolean	Not equal, treating null as a comparable value.	1 IS DISTINCT FROM NULL → t (rather than NULL) NULL IS DISTINCT FROM NULL → f (rather than NULL)
datatype IS NOT DISTINCT FROM datatype → boolean	Equal, treating null as a comparable value.	1 IS NOT DISTINCT FROM NULL → f (rather than NULL) NULL IS NOT DISTINCT FROM NULL → t (rather than NULL)
datatype IS NULL → boolean	Test whether value is null.	1.5 IS NULL → f
datatype IS NOT NULL → boolean	Test whether value is not null.	'null' IS NOT NULL → t
datatype ISNULL → boolean	Test whether value is null (nonstandard syntax).	
datatype NOTNULL → boolean	Test whether value is not null (nonstandard syntax).	
boolean IS TRUE → boolean	Test whether boolean expression yields true.	true IS TRUE → t NULL::boolean IS TRUE → f (rather than NULL)
boolean IS NOT TRUE → boolean	Test whether boolean expression yields false or unknown.	true IS NOT TRUE → f NULL::boolean IS NOT TRUE → t (rather than NULL)
boolean IS FALSE → boolean	Test whether boolean expression yields false.	true IS FALSE → f NULL::boolean IS FALSE → f (rather than NULL)
boolean IS NOT FALSE → boolean	Test whether boolean expression yields true or unknown.	true IS NOT FALSE → t NULL::boolean IS NOT FALSE → t (rather than NULL)
boolean IS UNKNOWN → boolean	Test whether boolean expression yields unknown.	true IS UNKNOWN → f NULL::boolean IS UNKNOWN → t (rather than NULL)
boolean IS NOT UNKNOWN → boolean	Test whether boolean expression yields true or false.	true IS NOT UNKNOWN → t NULL::boolean IS NOT UNKNOWN → f (rather than NULL)`

`BETWEEN` simplifies range tests. Endpoint values are treated as included.

```
a BETWEEN x AND y
a >= x AND a <= y
```

BETWEEN SYMMETRIC automatically swaps endpoint values if that to the left of AND is >= that to the right, so that a non-empty range is always implied.

```
mydb=> SELECT 3 BETWEEN 7 AND 2;
 ?column?
----------
 f
(1 row)

mydb=> SELECT 3 BETWEEN SYMMETRIC 7 AND 2;
 ?column?
----------
 t
(1 row)
```

> **NOTE:** The use of AND in BETWEEN syntax creates ambiguity with the use of AND as a logical operator, so only a limited set of expressions are allowed as the second argument of a BETWEEN clause.
>
> To write complex sub-expressions in BETWEEN, use ().

Ordinary comparison operators yield null ("unknown") when either input is null e.g. 7 = NULL and 7 <> NULL both yield null.

For non-null inputs, IS DISTINCT FROM is the same as <>. But if both inputs are null it returns false, and if only one input is null it returns true.

IS NOT DISTINCT FROM is similar to = for non-null inputs, but it returns true when both inputs are null, and false when only one input is null.

Use IS NULL and IS NOT NULL to check whether a value is null or not.

ISNULL and NOTNULL work too but are not standard.

If the *expression* is row-valued, IS NULL is true when the row expression itself is null or if all its fields are null; whereas IS NOT NULL is true when the row expression itself is non-null and all its fields are non-null.

Thus IS NULL and IS NOT NULL don't always return inverse results for row-valued expressions. A row-valued expression with both null and non-null fields returns false for both tests.

row IS DISTINCT FROM NULL and row IS NOT DISTINCT FROM NULL simply check the overal row value, with no additional checks on row fields.

Boolean values can be tested using predicates:

```
boolean_expression IS TRUE
boolean_expression IS NOT TRUE
boolean_expression IS FALSE
boolean_expression IS NOT FALSE
boolean_expression IS UNKNOWN
boolean_expression IS NOT UNKNOWN
```

These return true or false, never null.

Null input is treated as the logical value "unknown", so IS UNKNOWN and IS NOT UNKNOWN are effectively the same as IS NULL and IS NOT NULL when the input expression is boolean.

7.2.3 3. Comparison Functions

Function	Description	Example(s)
num_nonnulls (VARIADIC “any”) → integer	Returns the number of non-null arguments.	num_nonnulls(1, NULL, 2) → 2
num_nulls (VARIADIC “any”) → integer	Returns the number of null arguments.	num_nulls(1, NULL, 2) → 1

7.3 Mathematical Functions & Operators

7.3.1 1. Mathematical Operators

Operator	Description	Example(s)
numeric_type + numeric_type → numeric_type	Addition	2 + 3 → 5
+ numeric_type → numeric_type	Unary plus (no operation)	+ 3.5 → 3.5
numeric_type - numeric_type → numeric_type	Subtraction	2 - 3 → -1
- numeric_type → numeric_type	Negation	- (-4) → 4
numeric_type * numeric_type → numeric_type	Multiplication	2 * 3 → 6
numeric_type / numeric_type → numeric_type	Division (for integral types, division truncates the result towards zero)	5.0 / 2 → 2.5000000000000000 5 / 2 → 2 (-5) / 2 → -2
numeric_type % numeric_type → numeric_type	Modulo (remainder); available for smallint, integer, bigint, and numeric	5 % 4 → 1
numeric ^ numeric → numeric double precision ^ double precision → double precision	Exponentiation Unlike typical mathematical practice, multiple uses of ^ will associate left to right by default:	2 ^ 3 → 8 2 ^ 3 ^ 3 → 512 2 ^ (3 ^ 3) → 134217728
|/ double precision → double precision	Square root	|/ 25.0 → 5
||/ double precision → double precision	Cube root	||/ 64.0 → 4
@ numeric_type → numeric_type	Absolute value	@ -5.0 → 5
integral_type & integral_type → integral_type	Bitwise AND	91 & 15 → 11
integral_type | integral_type → integral_type	Bitwise OR	32 | 3 → 35
integral_type # integral_type → integral_type	Bitwise exclusive OR	17 # 5 → 20
~ integral_type → integral_type	Bitwise NOT	~1 → -2
integral_type << integer → integral_type	Bitwise shift left	1 << 4 → 16
integral_type >> integer → integral_type	Bitwise shift right	8 >> 2 → 2

Where *numeric_type* includes *integral_types*, `numeric`, `real` and `double precision`; and *integral_type* includes `smallint`, `integer` and `bigint`.

7.3.2 2. Mathematical Functions

Function
abs (numeric_type) → numeric_type
cbrt (double precision) → double
ceil (numeric) → numeric ceil (double precision) → double precision
ceiling (numeric) → numeric ceiling (double precision) → double precision
degrees (double precision) → double precision
div (y numeric, x numeric) → numeric
exp (numeric) → numeric exp (double precision) → double precision
factorial (bigint) → numeric
floor (numeric) → numeric floor (double precision) → double precision
gcd (numeric_type, numeric_type) → numeric_type
lcm (numeric_type, numeric_type) → numeric_type
ln (numeric) → numeric ln (double precision) → double precision
log (numeric) → numeric log (double precision) → double precision
log10 (numeric) → numeric log10 (double precision) → double precision
log (b numeric, x numeric) → numeric
min_scale (numeric) → integer
mod (y numeric_type, x numeric_type) → numeric_type
pi ( ) → double precision
power (a numeric, b numeric) → numeric power (a double precision, b double precision) → double precision
radians (double precision) → double precision
round (numeric) → numeric round (double precision) → double precision
round (v numeric, s integer) → numeric
scale (numeric) → integer
sign (numeric) → numeric sign (double precision) → double precision `
sqrt (numeric) → numeric sqrt (double precision) → double precision
trim_scale (numeric) → numeric
trunc (numeric) → numeric trunc (double precision) → double precision
trunc (v numeric, s integer) → numeric
width_bucket (operand numeric, low numeric, high numeric, count integer) → integer width_bucket (operand double precision, low
width_bucket (operand anycompatible, thresholds anycompatiblearray) → integer

Functions working with `double precision` data are mostly implemented on top of the host system's *C library*, so accuracy and behavior in boundary cases can vary depending on the host system.

7.3.3 3. Random Functions

Function	Description	Example(s)
random ( ) → double precision	Returns a random value in the range 0.0 <= x < 1.0	random() → 0.897124072839091
setseed (double precision) → void	Sets the seed for subsequent random() calls; argument must be between -1.0 and 1.0, inclusive	setseed(0.12345)

7.3.4 4. Trigonometric Functions

Function	Description	Example(s)
acos (double precision) → double precision	Inverse cosine, result in radians	acos(1) → 0
acosd (double precision) → double precision	Inverse cosine, result in degrees	acosd(0.5) → 60
asin (double precision) → double precision	Inverse sine, result in radians	asin(1) → 1.5707963267948966
asind (double precision) → double precision	Inverse sine, result in degrees	asind(0.5) → 30
atan (double precision) → double precision	Inverse tangent, result in radians	atan(1) → 0.7853981633974483
atand (double precision) → double precision	Inverse tangent, result in degrees	atand(1) → 45
atan2 (y double precision, x double precision) → double precision	Inverse tangent of y/x, result in radians	atan2(1, 0) → 1.5707963267948966
atan2d (y double precision, x double precision) → double precision	Inverse tangent of y/x, result in degrees	atan2d(1, 0) → 90
cos (double precision) → double precision	Cosine, argument in radians	cos(0) → 1
cosd (double precision) → double precision	Cosine, argument in degrees	cosd(60) → 0.5
cot (double precision) → double precision	Cotangent, argument in radians	cot(0.5) → 1.830487721712452
cotd (double precision) → double precision	Cotangent, argument in degrees	cotd(45) → 1
sin (double precision) → double precision	Sine, argument in radians	sin(1) → 0.8414709848078965
sind (double precision) → double precision	Sine, argument in degrees	sind(30) → 0.5
tan (double precision) → double precision	Tangent, argument in radians	tan(1) → 1.5574077246549023
tand (double precision) → double precision	Tangent, argument in degrees	tand(45) → 1

7.3.5 5. Hyperbolic Functions

Function	Description	Example(s)
sinh (double precision) → double precision	Hyperbolic sine	sinh(1) → 1.1752011936438014
cosh (double precision) → double precision	Hyperbolic cosine	cosh(0) → 1
tanh (double precision) → double precision	Hyperbolic tangent	tanh(1) → 0.7615941559557649
asinh (double precision) → double precision	Inverse hyperbolic sine	asinh(1) → 0.881373587019543
acosh (double precision) → double precision	Inverse hyperbolic cosine	acosh(1) → 0
atanh (double precision) → double precision	Inverse hyperbolic tangent	atanh(0.5) → 0.5493061443340548

7.4 String Functions & Operators

Strings in this context include `char`, `varchar` and `text`. `char` will be converted to `text` before the function or operator is applied, so trailing spaces are stripped.

7.4.1 SQL String Functions & Operators

Function/Operator	Description	Example(s)
text || text → text	Concatenates the two strings.	'Post' || 'greSQL' → PostgreSQL
text || anynonarray → text anynonarray || text → text	Converts the non-string input to text, then concatenates the two strings. (The non-string input cannot be of an array type, because that would create ambiguity with the array || operators. If you want to concatenate an array's text equivalent, cast it to text explicitly.)	'Value: ' || 42 → Value: 42
text IS [NOT] [form] NORMALIZED → boolean	Checks whether the string is in the specified Unicode normalization form. The optional form key word specifies the form: NFC (the default), NFD, NFKC, or NFKD. This expression can only be used when the server encoding is UTF8. Note that checking for normalization using this expression is often faster than normalizing possibly already normalized strings.	U&'\0061\0308bc' IS NFD NORMALIZED → t
bit_length (text) → integer	Returns number of bits in the string (8 times the octet_length).	bit_length('jose') → 32
char_length (text) → integer character_length (text) → integer	Returns number of characters in the string.	char_length('josé') → 4
lower (text) → text	Converts the string to all lower case, according to the rules of the database's locale.	lower('TOM') → tom
normalize (text [, form]) → text	Converts the string to the specified Unicode normalization form. The optional form key word specifies the form: NFC (the default), NFD, NFKC, or NFKD. This function can only be used when the server encoding is UTF8.	normalize(U&'\0061\0308bc', NFC) → U&'\00E4bc'
octet_length (text) → integer	Returns number of bytes in the string.	octet_length('josé') → 5 (if server encoding is UTF8)
octet_length (character) → integer	Returns number of bytes in the string. Since this version of the function accepts type character directly, it will not strip trailing spaces.	octet_length('abc '::character(4)) → 4
overlay (string text PLACING newsubstring text FROM start integer [FOR count integer]) → text	Replaces the substring of string that starts at the start'th character and extends for count characters with newsubstring. If count is omitted, it defaults to the length of newsubstring.	overlay('Txxxxas' placing 'hom' from 2 for 4) → Thomas
position (substring text IN string text) → integer	Returns first starting index of the specified substring within string, or zero if it's not present.	position('om' in 'Thomas') → 3
substring (string text [FROM start integer] [FOR count integer]) → text	Extracts the substring of string starting at the start'th character if that is specified, and stopping after count characters if that is specified. Provide at least one of start and count.	substring('Thomas' from 2 for 3) → hom substring('Thomas' from 3) → omas substring('Thomas' for 2) → Th
substring (string text FROM pattern text) → text	Extracts the first substring matching POSIX regular expression.	substring('Thomas' from '...$') → mas
substring (string text SIMILAR pattern text ESCAPE escape text) → text substring (string text FROM pattern text FOR escape text) → text	Extracts the first substring matching SQL regular expression. The first form has been specified since SQL:2003; the second form was only in SQL:1999 and should be considered obsolete.	substring('Thomas' similar '%#"o_a#"_' escape '#') → oma
trim ([LEADING | TRAILING | BOTH] [characters text] FROM	Removes the longest string containing only characters in characters (a space by default) from the start, end, or both ends (BOTH is the default) of string.	trim(both 'xyz' from 'yxTomxx') → Tom

7.4.2 Additional String Functions

Function	Descript
ascii (text) → integer	Returns t
btrim (string text [, characters text]) → text	Removes
chr (integer) → text	Returns t
concat (val1 “any” [, val2 “any” [, . . .]]) → text	Concaten
concat_ws (sep text, val1 “any” [, val2 “any” [, . . .]]) → text	Concaten
format (formatstr text [, formatarg “any” [, . . .]]) → text	Formats a
initcap (text) → text	Converts
left (string text, n integer) → text	Returns f
length (text) → integer	Returns t
lpad (string text, length integer [, fill text]) → text	Extends t
ltrim (string text [, characters text]) → text	Removes
md5 (text) → text	Compute
parse_ident (qualified_identifier text [, strict_mode boolean DEFAULT true]) → text[]	Splits qua
pg_client_encoding ( ) → name	Returns c
quote_ident (text) → text	Returns t
quote_literal (text) → text	Returns t
quote_literal (anyelement) → text	Converts
quote_nullable (text) → text	Returns t
quote_nullable (anyelement) → text	Converts
regexp_match (string text, pattern text [, flags text]) → text[]	Returns c
regexp_matches (string text, pattern text [, flags text]) → setof text[]	Returns c
regexp_replace (string text, pattern text, replacement text [, flags text]) → text	Replaces
regexp_split_to_array (string text, pattern text [, flags text]) → text[]	Splits stri
regexp_split_to_table (string text, pattern text [, flags text]) → setof text	Splits stri
repeat (string text, number integer) → text	Repeats s
replace (string text, from text, to text) → text	Replaces
reverse (text) → text	Reverses
right (string text, n integer) → text	Returns l
rpad (string text, length integer [, fill text]) → text	Extends t
rtrim (string text [, characters text]) → text	Removes
split_part (string text, delimiter text, n integer) → text	Splits stri
strpos (string text, substring text) → integer	Returns f
substr (string text, start integer [, count integer]) → text	Extracts t
starts_with (string text, prefix text) → boolean	Returns t
string_to_array (string text, delimiter text [, null_string text]) → text[]	Splits the
string_to_table (string text, delimiter text [, null_string text]) → setof text	Splits the
to_ascii (string text) → text to_ascii (string text, encoding name) → text to_ascii (string text, encoding integer) → text	Converts
to_hex (integer) → text to_hex (bigint) → text	Converts
translate (string text, from text, to text) → text	Replaces
unistr (text) → text	Evaluate

`concat`, `concat_ws` and `format` are *variadic*, so you can pass the values to be concatenated/formatted as an array marked with the `VARIADIC` keyword.

The array’s elements will be treated as separate ordinary arguments.

7.4.3 format

Produces output as specified by a *format string*, in likeness to C's `sprintf` function.

```
format(formatstr text [, formatarg "any" [, ...] ])
--e.g format('Hi %s', 'there!')
```

`formatstr` specifies how output should be formatted.

Each `formatarg` is converted to text, and then inserted into the result string as stipulated by *format specifiers*.

Format Specifiers

```
%[position][flags][width]type
```

- **position:** (optional)
  - `n$` where n is the index of the argument to print.
  - 1 refers to the first arg after *formatstr*, and so on.
  - If omitted, args are used in sequence.
- **flags:** (optional)
  - `-` causes output to be left-justified, but only if *width* is also specified.
- **width:** (optional)
  - `n`, `-n`, `*`(use next function arg as the width) or `*n$`(use *n*th function arg as the width).
  - Specifies the minimum number of characters to use to display the format specifier's output.
  - Output is left/right padded depending on the `-` flag.
  - Very small widths are ignored; output is not truncated.
- **type:** (required)
  - `s`(string), `I`(SQL identifier), `L`(SQL literal).

`%%` may be used to output a literal `%`

```
SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');   -- Testing three,␣
↪two, one
SELECT format('|%*2$s|', 'foo', 10, 'bar');      -- |       bar|
SELECT format('|%1$*2$s|', 'foo', 10, 'bar');    -- |       foo|
SELECT format('|%-*s|', -10, 'foo');     -- |foo       |
```

Unlike `sprintf`, `format`:

- allows format specifiers with and without *position* fields in the same format string.

  A format specifier without a position field always uses the next argument after the last argument consumed.
- doesn't require all function args to be used in the format string.

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');  -- Testing three, two,␣
↪three
```