PostgreSQL – Create Table with Index

Create Table with Index in PostgreSQL Database

In PostgreSQL, indexes are used to improve the performance of data retrieval operations by reducing the time required to locate specific rows in a table. You can create indexes on columns to optimize queries, and you can define them at the time of table creation or later using the `CREATE INDEX` command.

In this tutorial, we will explain how to create a table with indexes, demonstrates various scenarios, and provides detailed examples.

Basic Syntax

There are two main ways to create an index when creating a table:

- **Unique Constraints:** Automatically creates a unique index for the column(s).
- **Explicit Index Creation:** Create an index manually using the `CREATE INDEX` statement after the table is created.

Example Syntax:

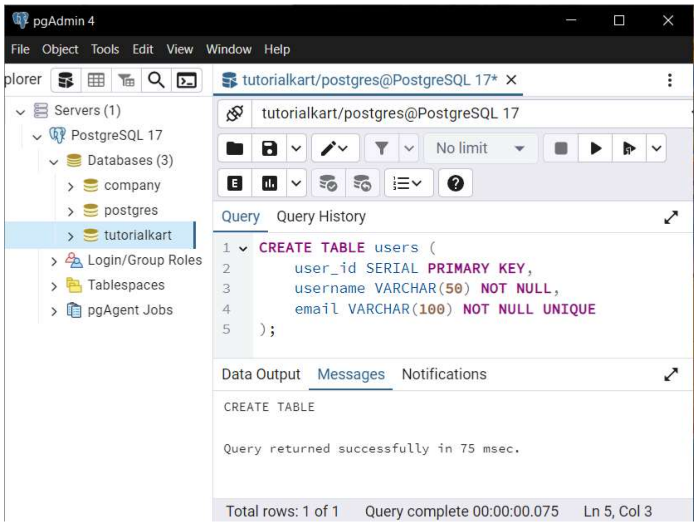
- Using a unique constraint: `CREATE TABLE ... UNIQUE(column_name);`
- Explicit index creation: `CREATE INDEX index_name ON table_name(column_name);`

Example 1: Create Table with Unique Index

Let's create a table named `users` where the `email` column has a unique constraint. This ensures that the `email` column is unique across all rows, and PostgreSQL automatically creates a unique index for this column.

```
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE
);
```

[What is Artificial Intelligence](#)



```
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE
);
```

```
CREATE TABLE

Query returned successfully in 75 msec.
```

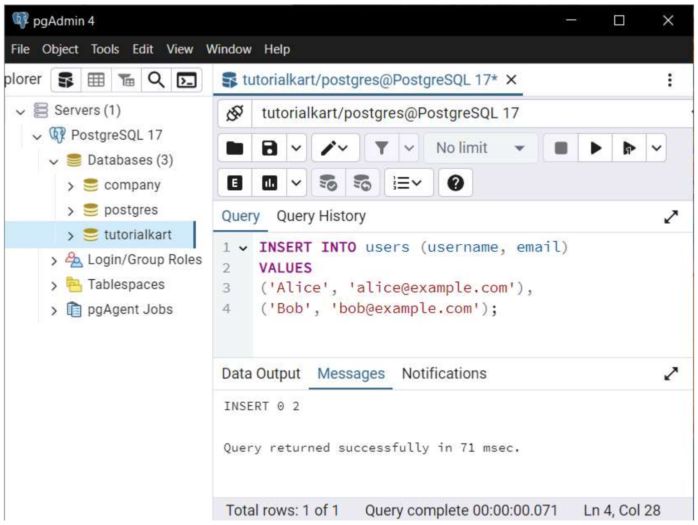
Total rows: 1 of 1 Query complete 00:00:00.075 Ln 5, Col 3

Explanation: The `UNIQUE` constraint on the `email` column automatically creates a unique index to enforce the constraint. This index ensures that no two rows have the same email address.

Insert sample data into the table:

</> Copy

```
INSERT INTO users (username, email)
VALUES
('Alice', 'alice@example.com'),
('Bob', 'bob@example.com');
```



```
INSERT INTO users (username, email)
VALUES
('Alice', 'alice@example.com'),
('Bob', 'bob@example.com');
```

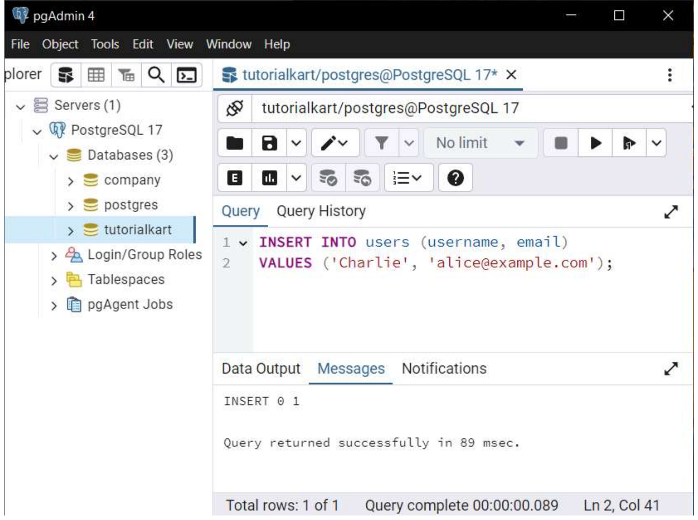
```
INSERT 0 2

Query returned successfully in 71 msec.
```

Total rows: 1 of 1 Query complete 00:00:00.071 Ln 4, Col 28

Attempt to insert a duplicate email:

</> Copy

```
INSERT INTO users (username, email)
VALUES ('Charlie', 'alice@example.com');
```

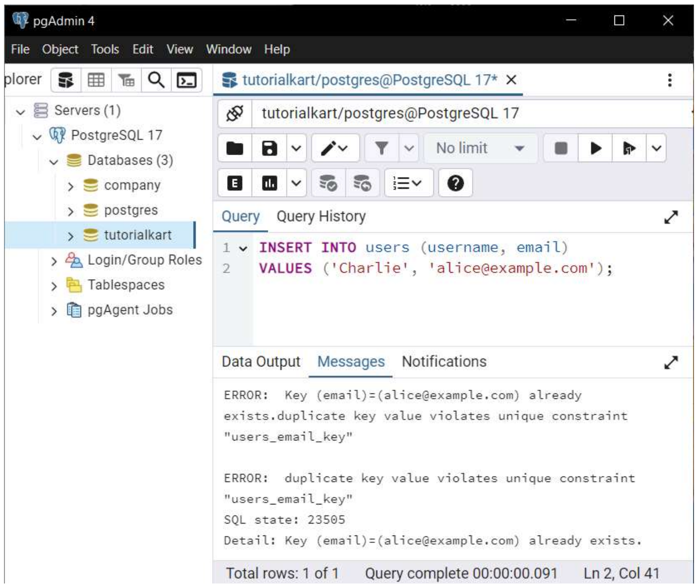


```
INSERT INTO users (username, email)
VALUES ('Charlie', 'alice@example.com');
```

```
INSERT 0 1

Query returned successfully in 89 msec.
```

Total rows: 1 of 1 Query complete 00:00:00.089 Ln 2, Col 41

Result: PostgreSQL will throw an error because the `email` column must be unique:



```
INSERT INTO users (username, email)
VALUES ('Charlie', 'alice@example.com');
```

```
ERROR:  Key (email)=(alice@example.com) already
exists.duplicate key value violates unique constraint
"users_email_key"

ERROR:  duplicate key value violates unique constraint
"users_email_key"
SQL state: 23505
Detail: Key (email)=(alice@example.com) already exists.
```

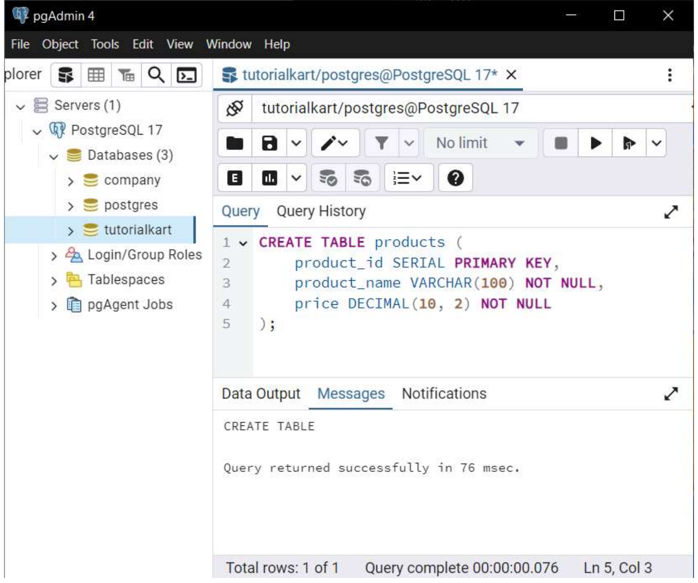
Total rows: 1 of 1 Query complete 00:00:00.091 Ln 2, Col 41

Example 2: Explicit Index Creation

Sometimes, you may want to create a non-unique index for faster query performance. Let's create a table named `products`, and then manually add an index on the `price` column.

</> Copy

```
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    price DECIMAL(10, 2) NOT NULL
);
```



```
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    price DECIMAL(10, 2) NOT NULL
);
```

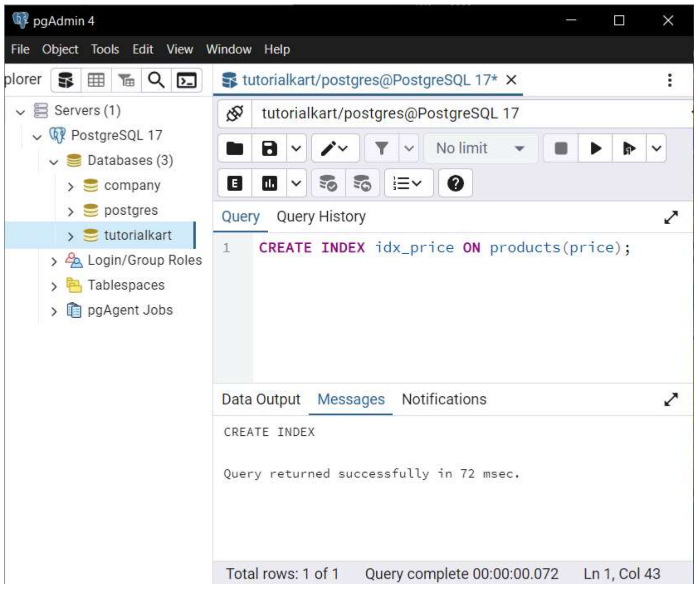
```
CREATE TABLE

Query returned successfully in 76 msec.
```

Total rows: 1 of 1 Query complete 00:00:00.076 Ln 5, Col 3

Create an index on the `price` column:

</> Copy

```
CREATE INDEX idx_price ON products(price);
```



```
CREATE INDEX idx_price ON products(price);
```

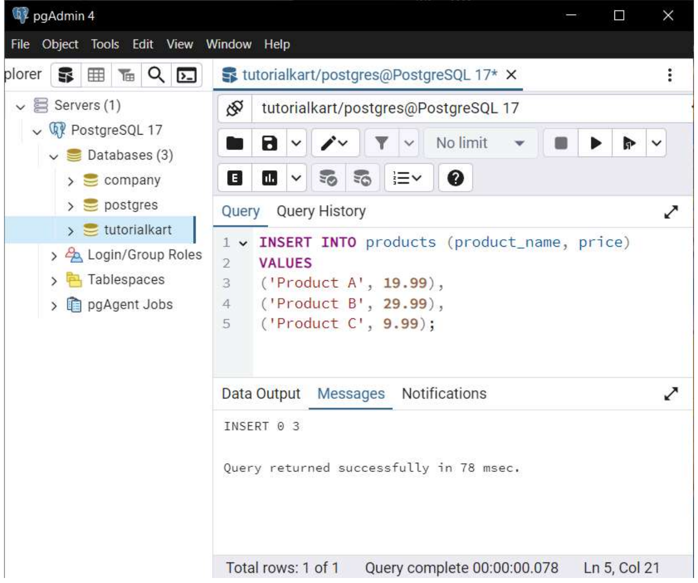
```
CREATE INDEX

Query returned successfully in 72 msec.
```

Total rows: 1 of 1 Query complete 00:00:00.072 Ln 1, Col 43

Insert sample data into the `products` table:

</> Copy

```
INSERT INTO products (product_name, price)
VALUES
('Product A', 19.99),
('Product B', 29.99),
('Product C', 9.99);
```



```
INSERT INTO products (product_name, price)
VALUES
('Product A', 19.99),
('Product B', 29.99),
('Product C', 9.99);
```

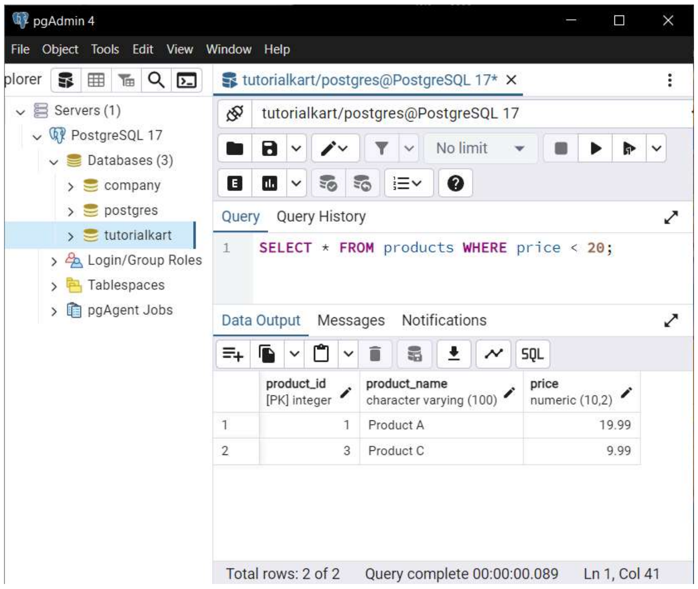
```
INSERT 0 3

Query returned successfully in 78 msec.
```

Total rows: 1 of 1 Query complete 00:00:00.078 Ln 5, Col 21

Query the table and filter results based on the **`price`** column:

</> Copy

```
SELECT * FROM products WHERE price < 20;
```



```
SELECT * FROM products WHERE price < 20;
```

	product_id [PK] integer	product_name character varying (100)	price numeric (10,2)
1	1	Product A	19.99
2	3	Product C	9.99

Total rows: 2 of 2 Query complete 00:00:00.089 Ln 1, Col 41

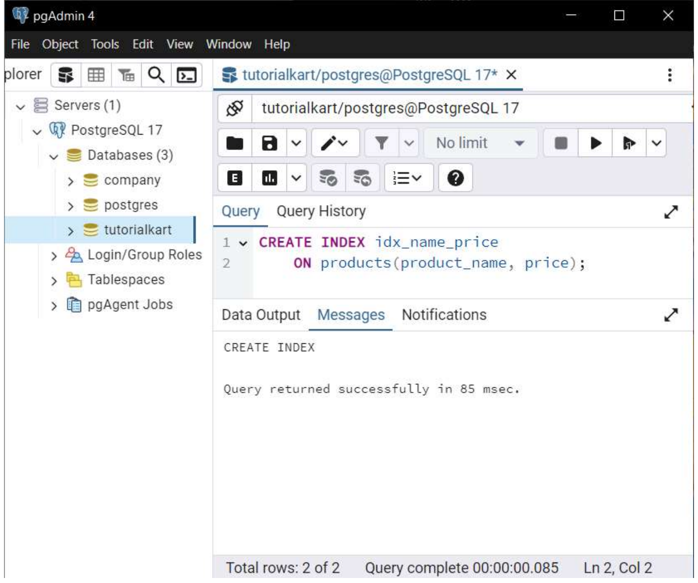
Explanation: The manually created index on the **`price`** column improves the performance of queries filtering on this column, as PostgreSQL uses the index to speed up the search.

Example 3: Composite Index

You can create a composite index on multiple columns to optimize queries that filter or sort by multiple criteria. Let's extend the **`products`** table and create a composite index on the **`product_name`** and **`price`** columns.

</> Copy

```
CREATE INDEX idx_name_price ON products(product_name, price)
```



```
CREATE INDEX idx_name_price
    ON products(product_name, price);
```

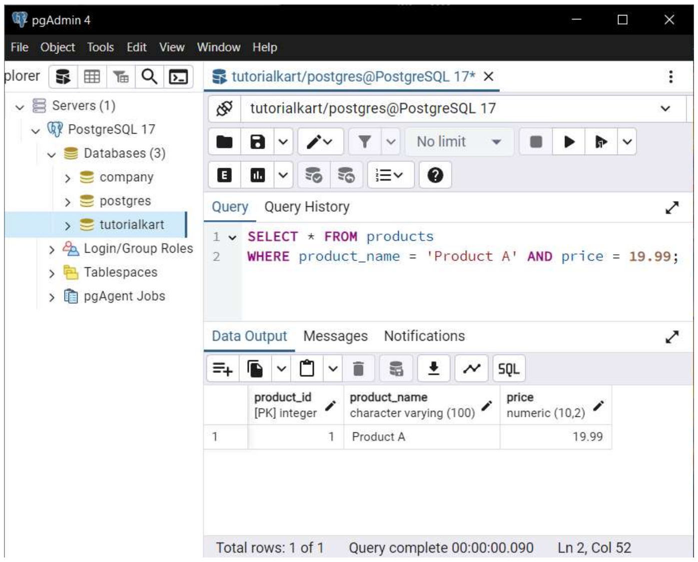
```
CREATE INDEX

Query returned successfully in 85 msec.
```

Total rows: 2 of 2   Query complete 00:00:00.085   Ln 2, Col 2

Query the table using both columns:

</>   Copy

```
SELECT * FROM products
WHERE product_name = 'Product A' AND price = 19.99;
```



```
SELECT * FROM products
WHERE product_name = 'Product A' AND price = 19.99;
```

	product_id [PK] integer	product_name character varying (100)	price numeric (10,2)
1	1	Product A	19.99

Total rows: 1 of 1   Query complete 00:00:00.090   Ln 2, Col 52

Explanation: The composite index improves the performance of queries that use both `product_name` and `price` in their filtering criteria. This is particularly useful for optimizing multi-column searches.

Conclusion

Indexes in PostgreSQL are a powerful tool for improving query performance. You can create them automatically using constraints like `UNIQUE`, or explicitly for specific optimization needs. By understanding the use of single-column indexes, composite indexes, and unique constraints, you can design your database to handle queries efficiently.

Top Walking Shoes for Seniors with Balance Issues

Popular Courses

SAP	CRM	Apache	Programming	Databases	Web & Server	Learning
SAP FI	Salesforce	Hadoop	C#	MongoDB	PHP	Maths
SAP CO	Salesforce Admin	Kafka Tutorial	C++	MySQL	HTML	Physics
SAP HR	Salesforce Developer	MXNet	Dart	PostgreSQL	CSS	Chemistry
SAP SD	Visualforce	OpenNLP	Golang		JavaScript	Units
SAP PS	Informatica	PDFBox	Java	**Mobile**	NGINX	Definitions
SAP ABAP		Spark Tutorial	Julia	Android Compose	NodeJS	Fashion
SAP MM	**SAP Resources**	Tomcat Tutorial	Kotlin	Flutter		General Knowledge
SAP PP	SAP Tables		Python	Kotlin Android	**Testing**	How-To
SAP PM	SAP Tcodes	**GUI**	R	SwiftUI	Selenium Java	
SAP GRC		JavaFX	Scala			
		Python Tkinter	Swift	**Linux**		
			TypeScript	Bash Script		