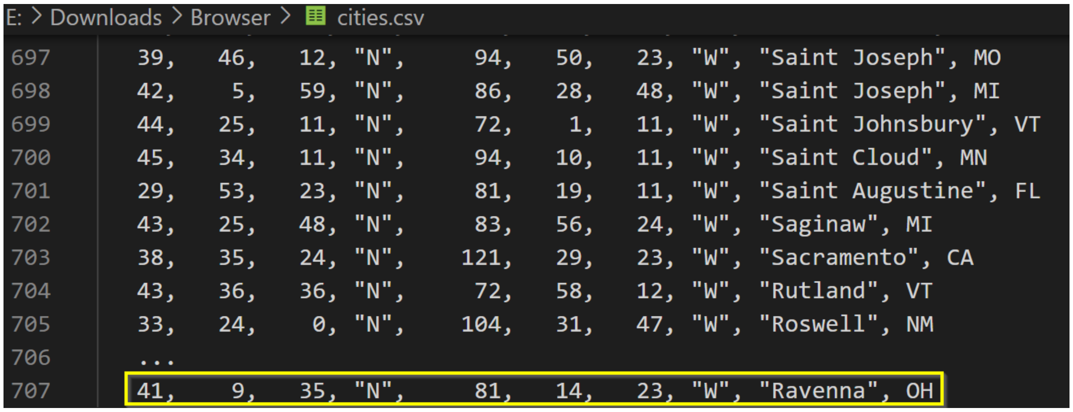
A PRIMER FOR SIMPLE DATABASES AND SQL

INTRODUCTION

Have you ever wondered how your phone knows the weather forecast for the next week? or Why is this website suggesting the same product that you were looking at last month? The short answer to these questions is **databases**. Data is the centerpiece of almost every application, and data persisting and querying is an art.

You are already familiar with storing text data in a file. Even though appending text to a file is not expensive, doing something more complex (such as making changes in the middle of a file) is not always viable performance-wise. Even something as simple as looking for a certain line would mean a scan of every line, and that is simply not feasible in many cases:



```
697    39,   46,   12, "N",     94,   50,   23, "W", "Saint Joseph", MO
698    42,    5,   59, "N",     86,   28,   48, "W", "Saint Joseph", MI
699    44,   25,   11, "N",     72,    1,   11, "W", "Saint Johnsbury", VT
700    45,   34,   11, "N",     94,   10,   11, "W", "Saint Cloud", MN
701    29,   53,   23, "N",     81,   19,   11, "W", "Saint Augustine", FL
702    43,   25,   48, "N",     83,   56,   24, "W", "Saginaw", MI
703    38,   35,   24, "N",    121,   29,   23, "W", "Sacramento", CA
704    43,   36,   36, "N",     72,   58,   12, "W", "Rutland", VT
705    33,   24,    0, "N",    104,   31,   47, "W", "Roswell", NM
706    ...
707    41,    9,   35, "N",     81,   14,   23, "W", "Ravenna", OH
```

Figure 1: Scanning a populated database line-by-line is tedious work

Consider the preceding example. If the record required is **`Ravenna`**, you won't find it until you go through 706 lines in the file. That means performing 706 comparison operations. However, if you were to use a database, you could perform this same task (in other words, accessing the required data) with only a few operations, which is clearly much more efficient. Likewise, if you were to insert a new record that started with, say, 'M' to a list in alphabetical order and went to do this manually, you would need to move roughly half the lines forward to make room; that's quite a time commitment for such a seemingly simple task.

Databases solve such problems by trying to add structure to the data being stored. Just like with classes, you can segregate different kinds of data into separate sections so that data that is similar can be accessed efficiently. Moreover, data often comes with a set of relations between its components. For example, a factory may produce many items, but these items may be made in only one factory. Such relations not only help optimize the retrieval of relevant data but also allow you to add validation rules, simplifying how you reason about data.

The first kind of database (and the most popular by far) is **relational database**. The kind of data that businesses use is usually structured and interconnected, with relations existing between various components; relational databases are meant for such purposes. **Structured Query Language** (**SQL**) is used to build such databases and then add, remove, update, or query data from them. Examples of SQL databases are SQL Server, SQLite, MySQL, Oracle, and PostgreSQL. Contrastingly, sometimes a database is semi-structured and constantly changing—not just by an increase in volume for already existing fields but also due to more fields being added to it. Such a database is called a document or **NoSQL** database. Examples of such databases are MongoDB, RavenDB, MariaDB, Cassandra, and CosmosDB.

Some databases are accessed publicly across networks, while others are only local. An effective example of a local database is SQLite, and it's what often holds the data in your mobile device. SQLite is serverless, meaning it doesn't require a fancy configuration or another server process to run.

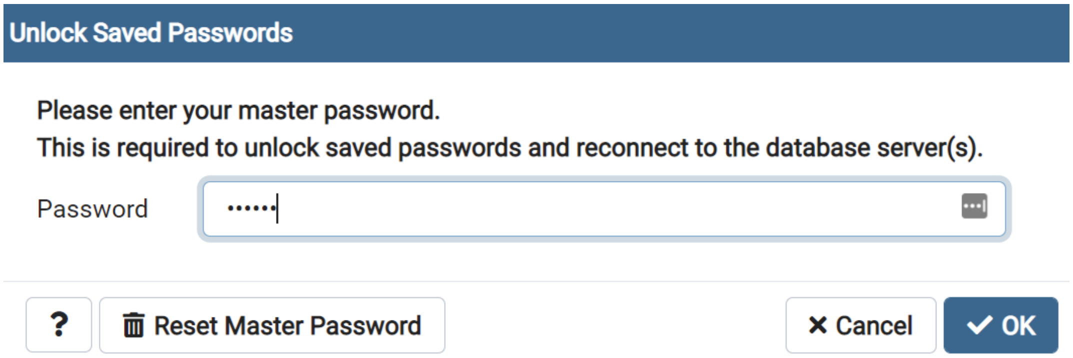
However, in this age of the internet, most databases need to be accessed remotely. Multiple processes should be able to consume the database simultaneously, and you don't want to allow inconsistencies or accidental bugs through. The database needs to be secure and trusted.

PostgreSQL Server is one such software. By itself, it's just a service running in the background, hosting other databases. To manage this service, you will need another piece of software. For PostgreSQL Server, one such tool is **pgAdmin**. In this chapter, you will learn how you can use PostgreSQL for database operations such as retrieving, updating, and deleting data. You will work with a real-world dataset and apply your database design skills to solve various business use cases. Learning how to work with databases is a cornerstone of becoming a professional software developer. The knowledge of SQL that you will get in this chapter is not PostgreSQL-specific; it is mostly transferable to all other dialects of SQL.

To learn the concepts taught in this chapter successfully, please ensure that you have installed pgAdmin and PostgreSQL Server. You can refer to the detailed steps in the *Preface* for these installations. Having grasped the basics of an SQL database, move on to connect to the SQL database PostgreSQL.

CONNECTING TO SQL SERVER

Once you open pgAdmin, you will see a browser window that requires you to enter the master password (which you set during the installation of PostgreSQL):



Unlock Saved Passwords

Please enter your master password.
This is required to unlock saved passwords and reconnect to the database server(s).

Password ••••••

Figure 2: PgAdmin connection to a local database using the master password

Once you have logged in with your password, you will see a collapsed server listing, as shown in *Figure 3*. When you first start using PostgreSQL, you will have only one server in the list (your local machine), but you can add more servers as you continue working with it. Expand **`Servers`**, and then expand **`Databases`**. You should see one database on your screen:

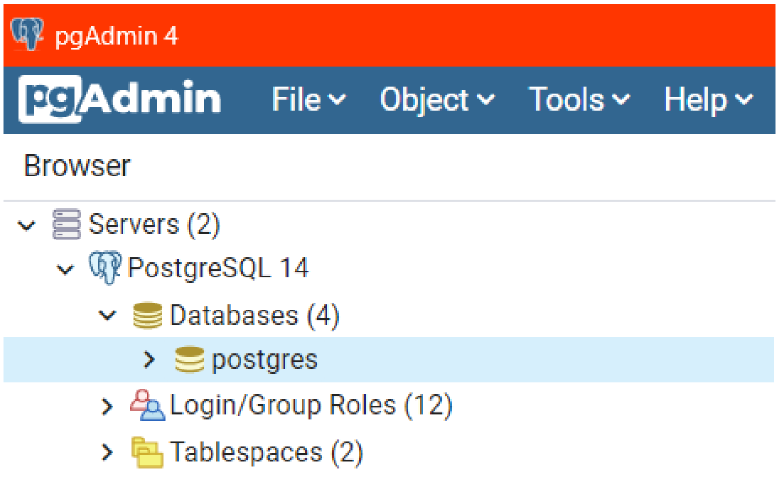


Figure 3: pgAdmin tool with a default database

For now, it is not very useful, because all you have is one default database. Let's create a new database.

INSTALLING A NEW DATABASE

You will now create a new database called **`Adventureworks`**. This is an example database used for both the reference chapter and *Chapter 6*, *A-Entity Framework with SQL Server*.

> **NOTE**
>
> You can also create a Database using PostgreSQL by clicking **`Databases`** and choosing **`Create`**. When the **`Create-Database`** dialog box gets displayed, type the Database name. However, here you will use the command line as it is needed for the installation script. Make sure you follow all the steps mentioned in the Preface under the *PostgreSQL Installation for Windows* section and the addition of environment variables.

Perform the following steps to do so:

1. Clone https://packt.link/sezEm.
2. Open the command line and navigate to https://packt.link/I1KW3 (local files equivalent).
3. Then create an empty **`Adventureworks`** database by running the following command in the console:

```
psql -U postgres -c "CREATE DATABASE \"Adventureworks\";"
```

4. Create tables and populate them with data using the installation script.
5. Run the following command pointing to the installation script:

```
psql -d Adventureworks -f install.sql -U postgres
```

It's worth mentioning that not every database uses SQL. In general, they have their individual approaches to it. PostgreSQL Server uses PSQL. However, in the examples that follow, you will not use PostgreSQL-specific features. So, most queries written in PSQL should apply to all SQL databases. Up next, you will explore an example database structure.

DATABASE STRUCTURE

A server consists of one or more databases. Data is not stored directly on a server but rather inside a database. If you expand the database you have created (or imported) with the sample data, you should see something like the following:

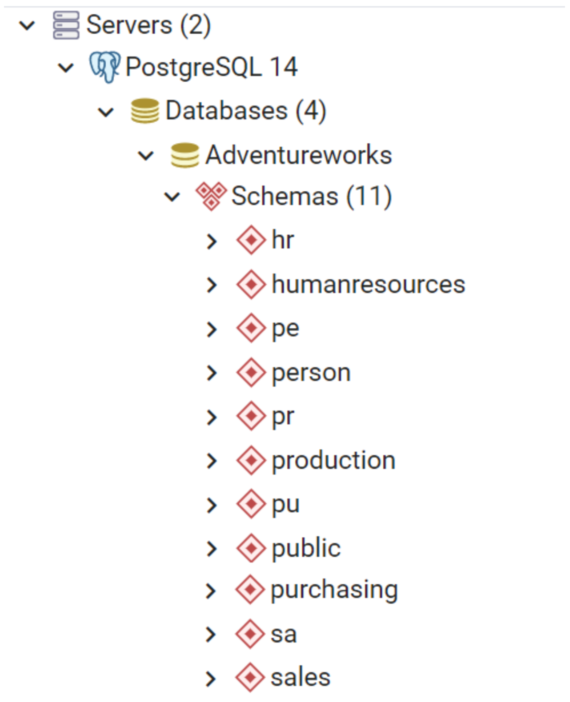


Figure 4: Database structure inside the database browser (image simplified for brevity)

The key thing to notice here is that there are **11 schemas**. A schema is a logical grouping of related data. A single schema contains various logically related objects to either store data or helps store it.

> **NOTE**
>
> The schemas shown in *Figure 4* are **11** because they show the post-installation script result.

The most important part of a relational database is a **table**. This is where all the data is stored. In a nutshell, you can consider a database to be a collection of tables. A simple database can have just a few tables, but for bigger applications, there may be hundreds or thousands of such tables. To see this more clearly, expand the **`production`** schema, and then expand **`Tables`**, as shown in the following figure:

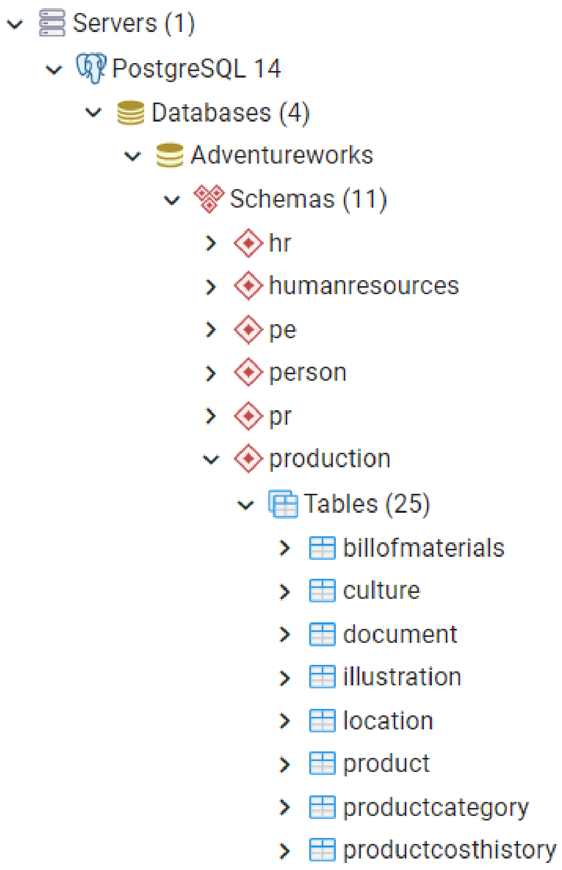


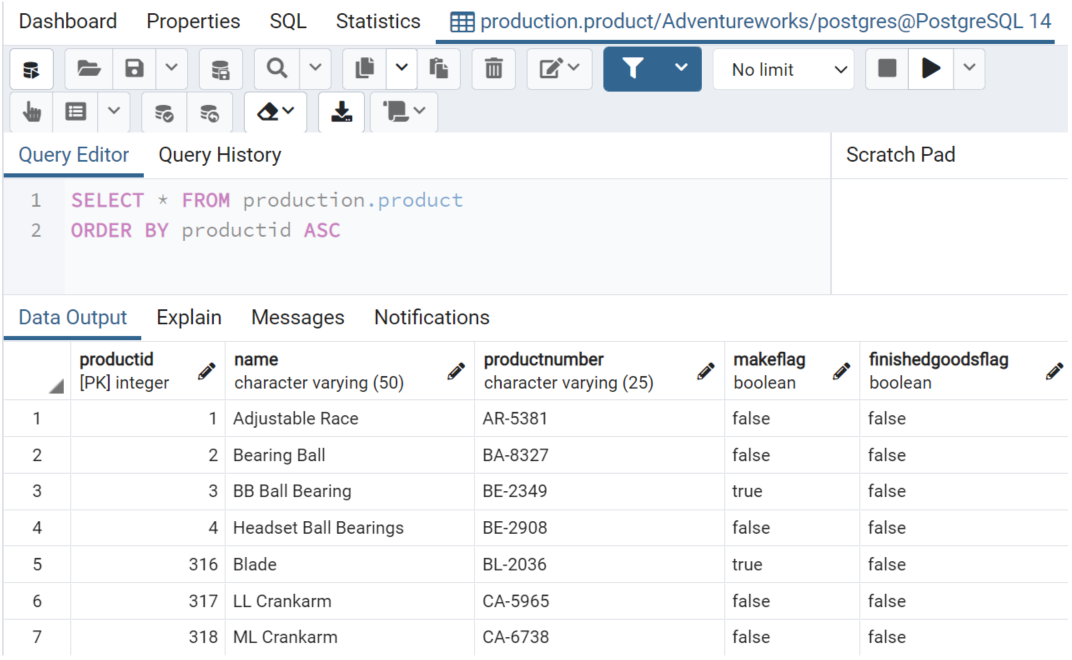
Figure 5: The production schema, its objects, and its tables

You will notice that the **`production`** schema has a lot inside it. However, the important thing to notice here is that there are **`23`** tables.

Databases are optimized for saving and looking up data. Internally, data is balanced in a binary tree or another type of data structure. Optimized data storage doesn't happen by default, however, and you should implement the basic steps for achieving that, as you will see in the next section.

TABLE STRUCTURE

If you right-click on the **`product`** table and select **`View/Edit Data`**, followed by **`All Rows`**, a new window will open in two parts. The bottom part is a table, as shown in *Figure 6*:



```
SELECT * FROM production.product
ORDER BY productid ASC
```

	productid [PK] integer	name character varying (50)	productnumber character varying (25)	makeflag boolean	finishedgoodsflag boolean
1	1	Adjustable Race	AR-5381	false	false
2	2	Bearing Ball	BA-8327	false	false
3	3	BB Ball Bearing	BE-2349	true	false
4	4	Headset Ball Bearings	BE-2908	false	false
5	316	Blade	BL-2036	true	false
6	317	LL Crankarm	CA-5965	false	false
7	318	ML Crankarm	CA-6738	false	false

Figure 6: The products table displaying rows and columns through PgAdmin

In this figure, you can see the **`product`** table. Every table has **columns** and **rows**. As you know, a table is like a blueprint for the data, just as a class is for objects in C#. A row is an instance of the data and represents what you get when the blueprint is applied to create a meaningful object. Every column represents a field and has a data type associated with it.

A row is a single record in a database—the smallest unit of data that makes sense by itself. Looking at any row, you should be able to either learn everything about the data right away or see hints (IDs to other tables and records) pertaining to the rest of the data.

The **data types** in a database are like primitive C# data types, but they are much more precise in terms of what you can store using them. For example, for fixed-sized characters, the **`char`** data type is used; but if the size of the characters is variable, you would use the **`varchar`** data type.

To make the product table, perform the following steps:

1. From **`pgAdmin4`**, expand the **`Adventureworks`** database.
2. Click on **`Schemas`**.
3. Right click **`Table`** and choose **`Create`**.
4. Type the name of the table as **`product`**.
5. Right click on **`product`**, choose **`Create`** and click **`Column`**.
6. In the **`General`** tab of the Create-Column dialog box, type the name of the column as **`productid`**.
7. Choose **`integer`** as its data type under the **`Definition`** tab and click **`Save`**.
8. Similarly make the other column names as shown in the table in *Figure 6*.

   If you look closely, you can see the underlying types under the column names displayed in the table in *Figure 6*. The **`product`** table branch under pgAdmin is shown in *Figure 7*:

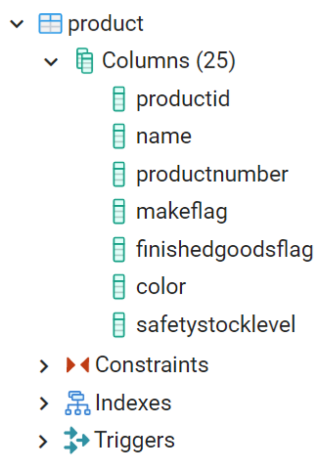


Figure 7: The product table showing columns, constraints, indexes, and triggers

Here, **`productid`** is an integer, analogous to **`int`** in C#, and **`makeflag`** is a Boolean, which is analogous to **`bool`** in C#. Some types (for example, **`char`**) have a length property associated with them. Text types usually have this because, if you don't specify the length, you reserve more space than needed. In addition to storing data, a table also includes other objects to make working with data efficient and consistent, as you'll see in the next sections.

KEYS

Under the columns, you can see the **`Constraints`** folder. Create two entries: **`product_pkey`** and **`product_manufacturerid_id`**. A key is used to identify a row or link one row with another. There are two types of keys: primary and foreign:

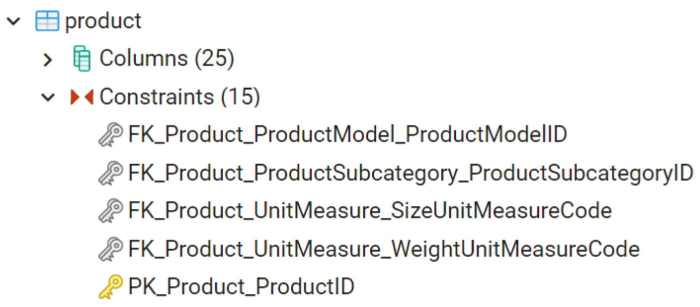

product
Columns (25)
Constraints (15)
FK_Product_ProductModel_ProductModelID
FK_Product_ProductSubcategory_ProductSubcategoryID
FK_Product_UnitMeasure_SizeUnitMeasureCode
FK_Product_UnitMeasure_WeightUnitMeasureCode
PK_Product_ProductID

Figure 8: Constraints of the product table showing PK_Product_ProductID primary key with foreign keys

Every table should have at least one **primary key**, which is used to identify a unique row. Any column, as long as it is unique, can be used as a primary key. For example, an email column could be used to identify people. However, that is not optimal because comparing text is slower than comparing numbers. So, a better choice usually is to have a dedicated ID column of a numeric type for the primary key. Here **`product_pkey`** is made the primary key.

There are two main choices of a primary key column data type: **`int`** or **`guid`**. Both are sound choices because generating a new ID is straightforward. It involves either incrementing the previous ID or generating a new **`guid`** key. Note that **`guid`** should be chosen only when multiple databases are involved. This is because incremented keys across databases are likely to overlap, but the nature of **`guid`** makes them unique, so there is a low risk of clashing primary keys. Naming an ID column is as simple as using **`id`** or prefixing it with a table name (**`productid`**). There aren't any strict rules for naming ID columns, but it helps to stick to conventional and consistent naming.

The other type of key is a **foreign key** (**FK**), which is used to reference a row from another table. If foreign keys did not exist, you would be forced to duplicate data across multiple rows. For example, consider the **`product`** table, where a product has a name, price, category, and model. Similarly, a model has a name, description, instructions, and more. If you stored 10 products of the same model, you would be forced to also duplicate that model's data, despite it being the same. In *Figure 8*, **`product_manufacturerid_id`** is the foreign key.

A foreign key of a table is used to make a link with another table. It allows you to easily remove data duplications. Instead of repeating columns with the same data in the same table, you can separate them into different tables, keeping only the differences in individual tables. The benefit when querying is that you will still get duplicate data. However, there will be no duplicates stored, thereby saving quite some space.

In the **`Adventureworks`** database, you have three key columns: **`productid`**, **`productcategoryid`**, and **`productmodelid`**. **`Productid`** identifies a unique product; **`productcategoryid`** and **`productmodelid`** link individual products with the category and model they belong to, respectively.

It is common to conflate the ID columns with keys. However, an ID is not always a key; it needs to be marked as such first. To inspect primary keys, right-click on a table and select **`Properties`**..., as shown in *Figure 9*:

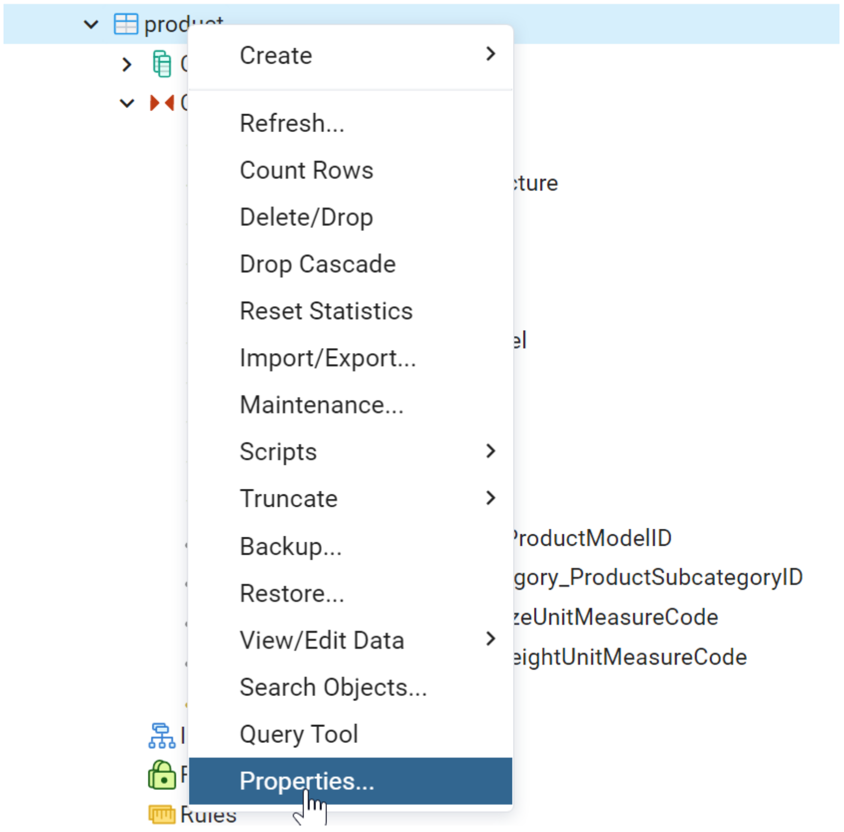
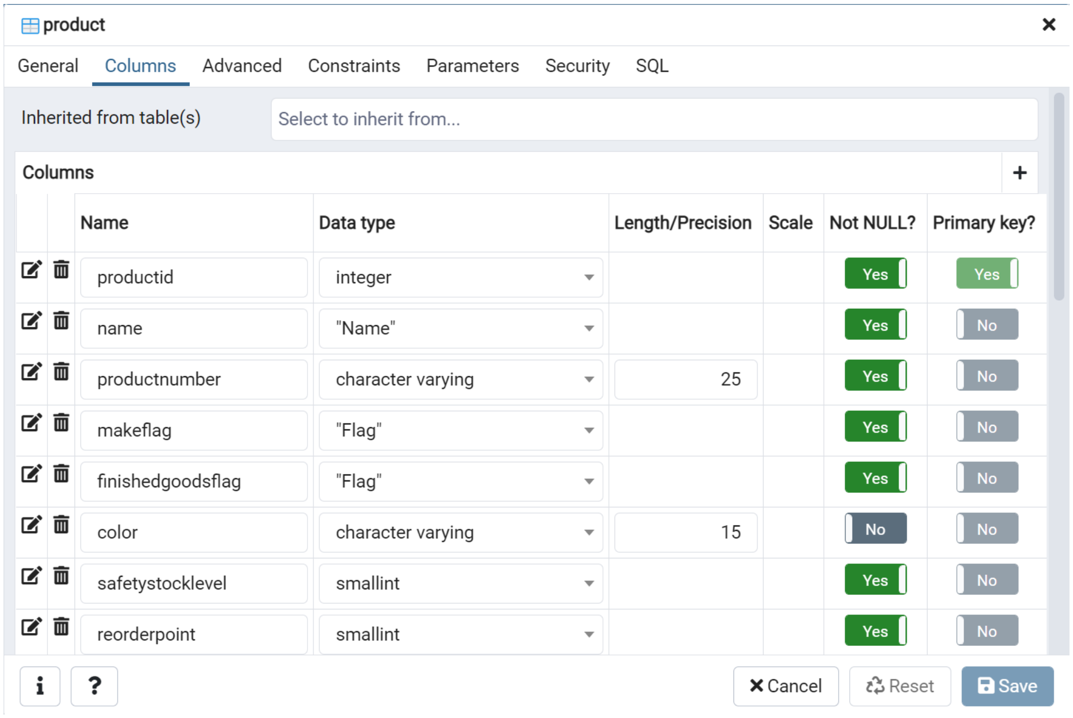


Figure 9: Selecting Properties to inspect primary keys in the product table

Then, navigate to the **`Columns`** tab. You will see that **`productid`** is flagged as a primary key:



Inherited from table(s): Select to inherit from...

Columns

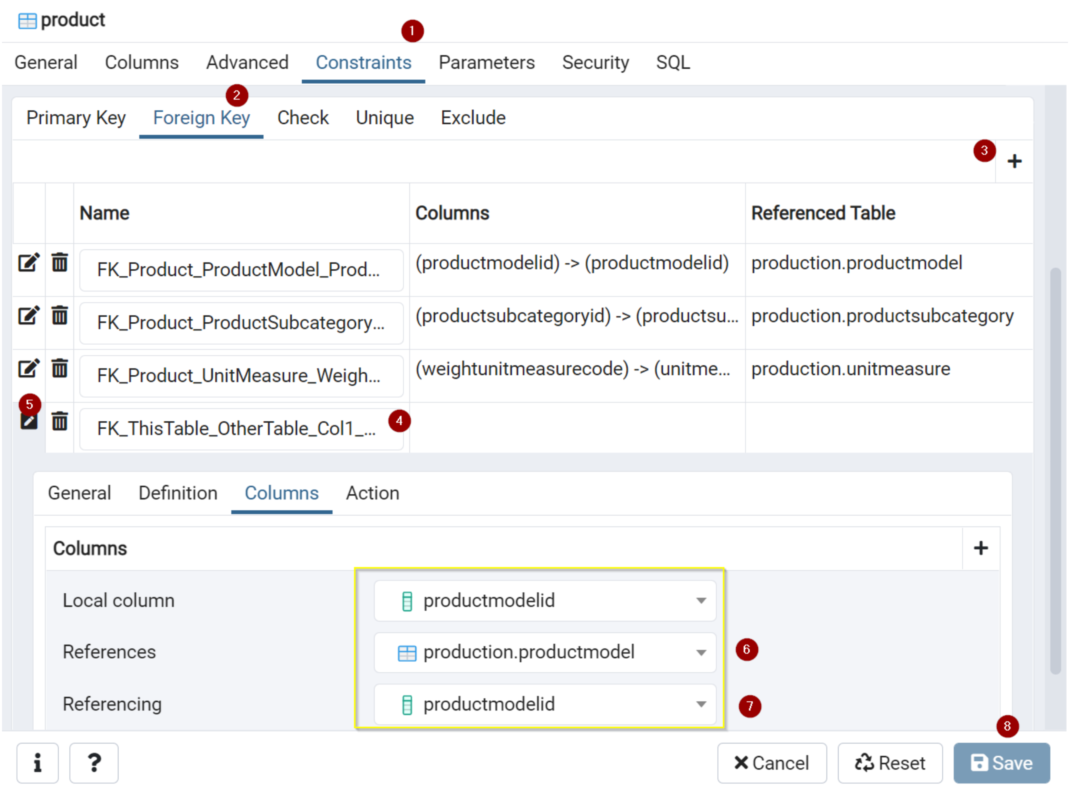
Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?
productid	integer			Yes	Yes
name	"Name"			Yes	No
productnumber	character varying	25		Yes	No
makeflag	"Flag"			Yes	No
finishedgoodsflag	"Flag"			Yes	No
color	character varying	15		No	No
safetystocklevel	smallint			Yes	No
reorderpoint	smallint			Yes	No

Figure 10: The product table properties with productid chosen as the primary key

To add a 1:n foreign keys relationship, perform the following steps:

1. Select **`Constraints`** (**1**) as shown in *Figure 11*, and then select **`Foreign Key`** (**2**).
2. Next, click on **`Add row`** button (**3**) and enter a new foreign key name (**4**).
3. Then click on the **`Edit row`** button (**5**).
4. Now, reference an outer table (**6**) and link it with an outer column on that table (**7**).

5. Click the **Add** button and finally the **Save** button (**8**):



Name	Columns	Referenced Table
FK_Product_ProductModel_Prod...	(productmodelid) -> (productmodelid)	production.productmodel
FK_Product_ProductSubcategory...	(productsubcategoryid) -> (productsu...	production.productsubcategory
FK_Product_UnitMeasure_Weigh...	(weightunitmeasurecode) -> (unitme...	production.unitmeasure
FK_ThisTable_OtherTable_Col1_...		

Columns	
Local column	productmodelid
References	production.productmodel
Referencing	productmodelid

Figure 11: Adding a 1:n foreign key relationship

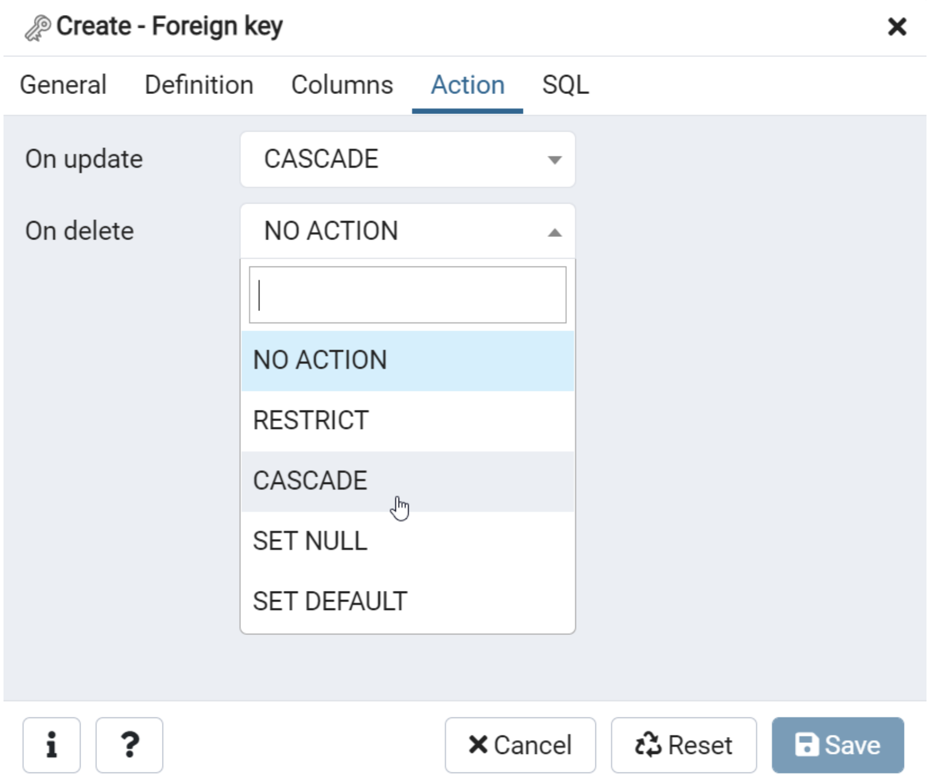
So, what happens when you delete or update records in a parent table? This is where you need to know about Cascade actions.

CASCADE ACTIONS

It's worth noting that deleting or updating data is not straightforward as, ideally, all the linked tables should be in sync as well. For example, consider that every row in a `Dogs` table has a related `Owner` row. In such a case, deleting the owner row should get rid of the dog rows as well. Such a requirement can be taken care of by the following steps:

1. Go to the **Action** tab in the **Foreign Key** Properties window.

2. Choose the **`CASCADE`** option beside the **`On delete`** action dropdown, as shown in *Figure 12*:



On update: CASCADE

On delete: NO ACTION

NO ACTION
RESTRICT
CASCADE
SET NULL
SET DEFAULT

Figure 12: Using the CASCADE option from On update and On delete action for product foreign key constraints

> **NOTE**
>
> You can only choose an action for a **new** FK constraint. For existing FK constraints, you will only have actions in read-only mode.

A big advantage of a SQL database over NoSQL is that you can use constraints, which are applied during write operations so that invalid data is not stored. This is where the concept of constraints needs to be uncovered.

Constraints

You are already familiar with primary and foreign key constraints. As you know, a primary key column must contain unique values (after all, it identifies a row uniquely). A foreign key constraint requires a primary key of that value to exist in the linked table.

In addition to these, the **`NotNull`** constraint prevents null values from being stored. The **`Default`** constraint assigns a default value when inserting a row where no value is provided for a column with that constraint. Another constraint is **`Unique`**. Similar to a primary key, it requires columns to be **unique**. However, unlike a primary key, it does not provide any identity to a row. There is also a **`Check`** constraint that verifies whether a column meets a certain condition (for example, a distance fewer than three meters).

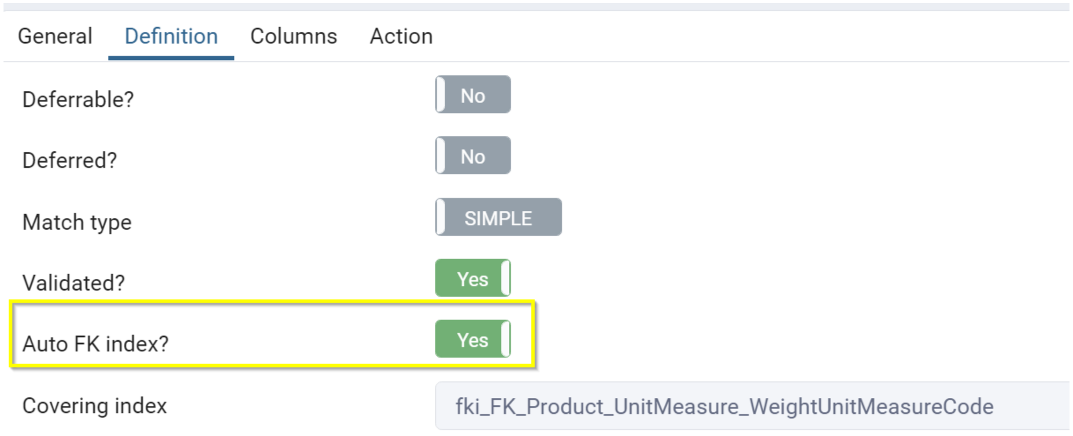
Let's move on from data validation to performance topic—indexes.

Indexes

The meaning of index for both databases and books is very similar. In a book, the index points to the right page. In a table, an index is used for the fast lookup of rows. Columns likely to be used in a search should be indexed. A **clustered index** is a kind of index that physically sorts a table (each write operation will insert a new row in a sorted position). A table can have only **one** clustered index. A effective example of that is a primary key, which is extremely likely to be used when querying data. Not all primary keys are clustered indexes, but that is the default choice.

Often, you will need to do a lookup based on criteria such as a product name or a person's age. If you care greatly about the speed of lookup, you should create an index on those columns. Such indexes are described as **non-clustered**. Multiple non-clustered indexes can exist, but they work slower than clustered indexes. You can also create a non-clustered index on the fly, using SQL. That is actually a common practice; however, the topic is outside the scope of this chapter.

PostgreSQL creates a clustered index by default for primary keys and unique constraints. However, no index is added for foreign keys by default. It is often needed for faster lookup; thus, in order to achieve that from the **`Table Properties`** window, go to the **`Definition`** tab. You can mark a foreign key as an index, as shown in *Figure 13*:



Deferrable?	No
Deferred?	No
Match type	SIMPLE
Validated?	Yes
Auto FK index?	Yes
Covering index	fki_FK_Product_UnitMeasure_WeightUnitMeasureCode

Figure 13: FK marked as an index for faster lookup

The next section discusses triggers which are actions required for adding, removing, or deleting data.

TRIGGERS

PostgreSQL Server can react to **events** – specifically, data modification events. Sometimes after adding, removing, or deleting data, you will also need to perform some action. Such an action is called a **trigger**.

In general, a great use case for triggers is auditing data. For instance, you may want to write everything that happened to data as-is, as a kind of a log for changes made on a row. Auditing a table usually gets rows inserted and not updated (because you want to insert new changes on data and not update existing ones). For example, for every change to a **`users`** table, a trigger could be created so that a new audit entry is inserted upon addition or modification of a user row.

NOTE

This topic is outside the scope of this chapter. If interested to know more about them, refer to https://www.postgresql.org/docs/9.1/sql-createtrigger.html.

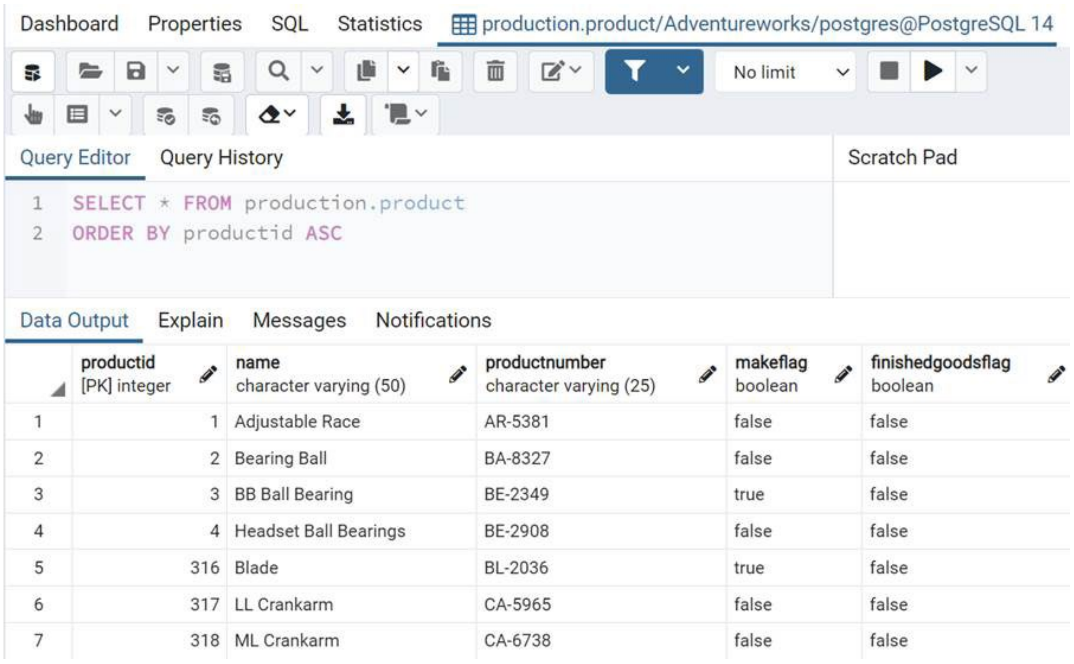
READING AND WRITING TO A DATABASE

So far, this chapter has mostly explored the structure of databases. Now it's time to look at what you can do with the data.

VIEWING DATA

To view all the data in a table, perform the following steps:

1. Right-click on the **`product`** table.
2. Select **`View/Edit Data`**.
3. Choose **`All Rows`**. You will observe a similar screen to *Figure 14*:



```
SELECT * FROM production.product
ORDER BY productid ASC
```

	productid [PK] integer	name character varying (50)	productnumber character varying (25)	makeflag boolean	finishedgoodsflag boolean
1	1	Adjustable Race	AR-5381	false	false
2	2	Bearing Ball	BA-8327	false	false
3	3	BB Ball Bearing	BE-2349	true	false
4	4	Headset Ball Bearings	BE-2908	false	false
5	316	Blade	BL-2036	true	false
6	317	LL Crankarm	CA-5965	false	false
7	318	ML Crankarm	CA-6738	false	false

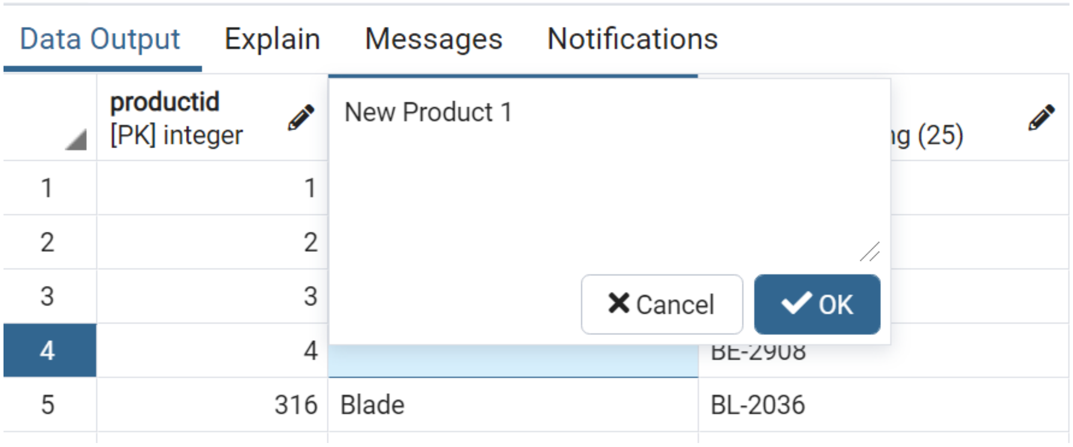
Figure 14: Generated SQL query that returns all data in a table

This time you'll explore the top part. The top part contains a generated SQL query that returns all the data in a table. The view that is generated at the bottom of the screen that shows the output is called the results view.

MODIFYING DATA

Changing the values of data, and adding or removing rows, is also very simple. To change a product name first, perform the following steps:

1. Double-click on any row of the **`productname`** column. A new window appears.
2. Type the new **`name`** for the product (*Figure 15*):



	productid [PK] integer		
1	1		
2	2		
3	3		
4	4		BE-2908
5	316	Blade	BL-2036

New Product 1

Figure 15: Changing row content of a table

3. After that, hit **OK** and press *F6*. You can alternatively click the **`Save Data Changes`** button (as shown in *Figure 16*):

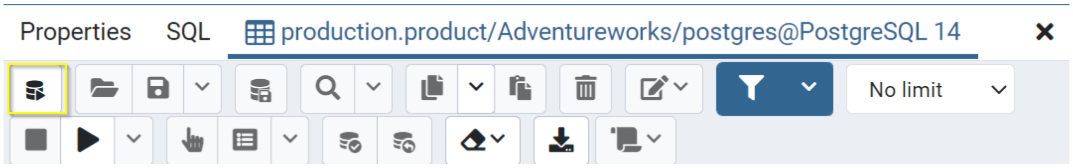
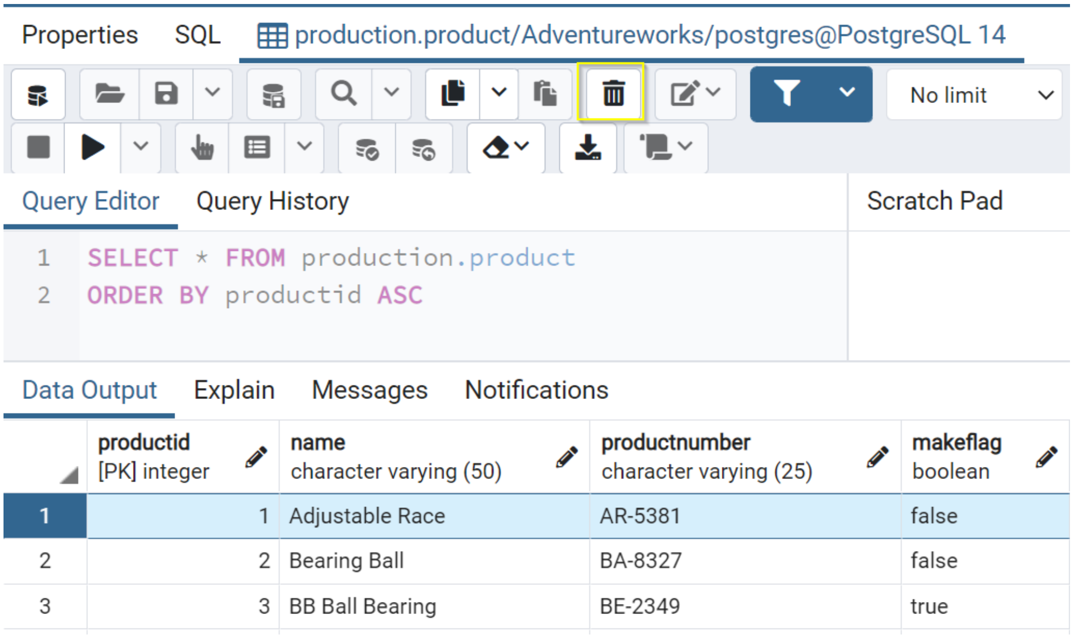


Figure 16: Saving any changes to data values with the Save Data Changes button

4. To add a new row, just type the values in the bottom-most row of the results view (the results view is the Data Output tab at the bottom side of the screen).

5. To remove a row, select the row by clicking the particular row number and hit the **`Delete`** button, as shown in *Figure 17*:



```
SELECT * FROM production.product
ORDER BY productid ASC
```

	productid [PK] integer	name character varying (50)	productnumber character varying (25)	makeflag boolean
1	1	Adjustable Race	AR-5381	false
2	2	Bearing Ball	BA-8327	false
3	3	BB Ball Bearing	BE-2349	true

Figure 17: Deleting a row with the Delete button

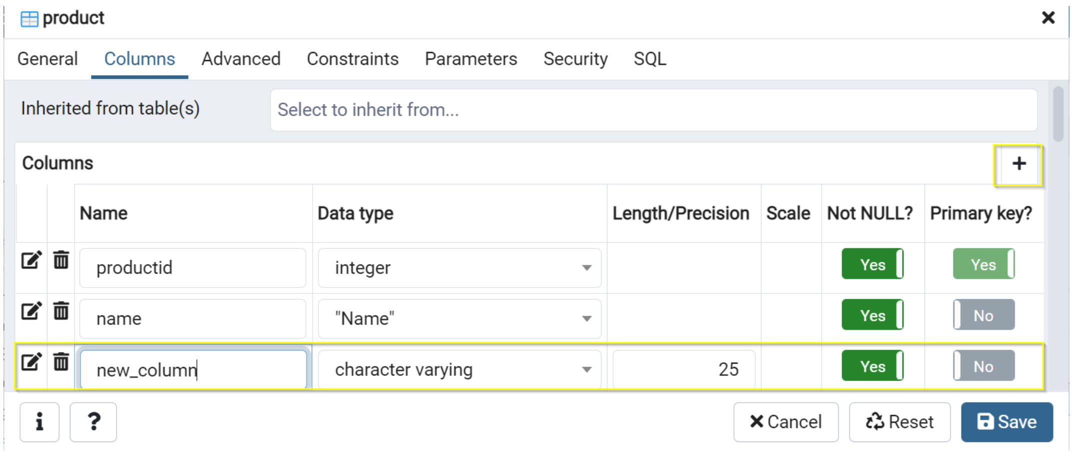
ADDING AND MODIFYING COLUMNS

To add new columns or change existing ones through pgAdmin, perform the following steps:

1. To view table properties view. For this, right-click a table and select **`Properties`**.

2. Then, go to **`Columns`**.

3. Click on the Add (**+**) button. A new column gets added:



Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?
productid	integer			Yes	Yes
name	"Name"			Yes	No
new_column	character varying	25		Yes	No

Figure 18: Adding a new column

4. Type the new column name.
5. Choose a **`Data type`**, **`Length/Precision`**,(how many max characters or digits can a column store) **`Scale`** (how many decimal places do numbers (only) contain).
6. Add simple constraints such as **`Not NULL`**.
7. Flag the column as a primary key, if required.
8. Click the **`Save`** button to save the newly added column.
9. Click on the **`View Data`** button to view the newly added column in the results view.

NEW TABLE

To create a new table, perform the following steps:

1. Expand any **`Schemas`**.
2. Right-click on **`Tables`**.
3. Then select **`Create`** and **`Table`** (*Figure 19*):

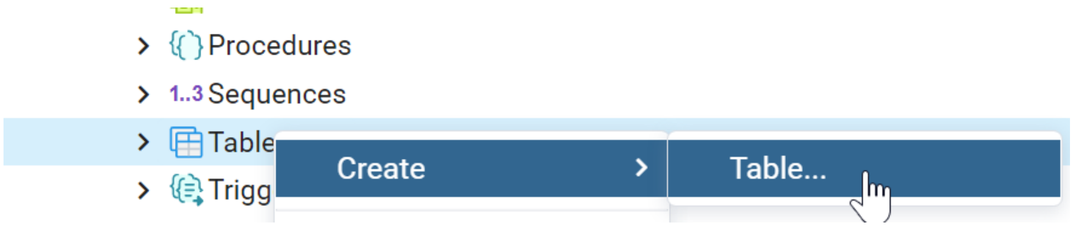
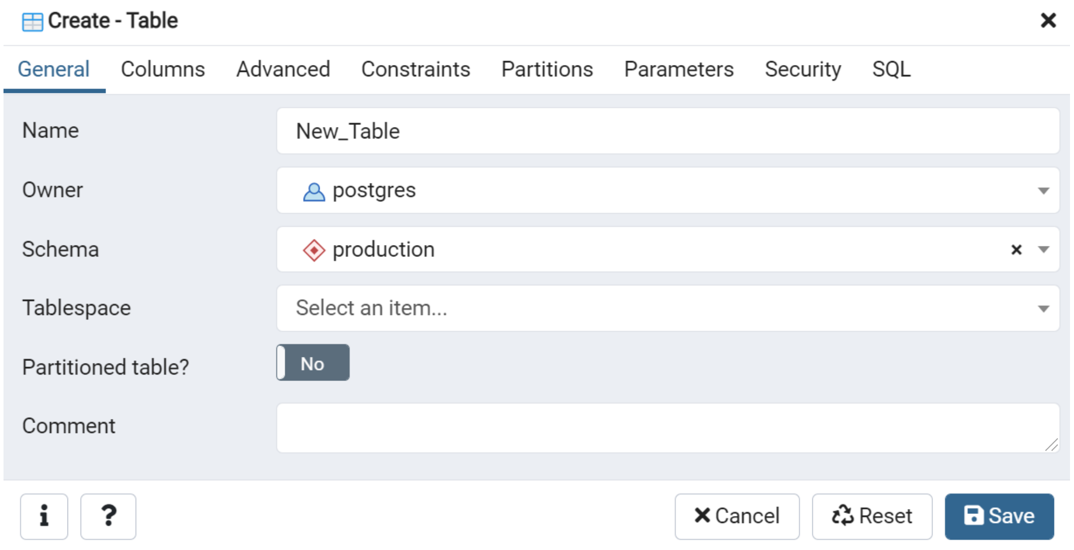


Figure 19: Creating a new table

You will be moved to an already familiar table properties view where you can name the table, add columns, and do much more:



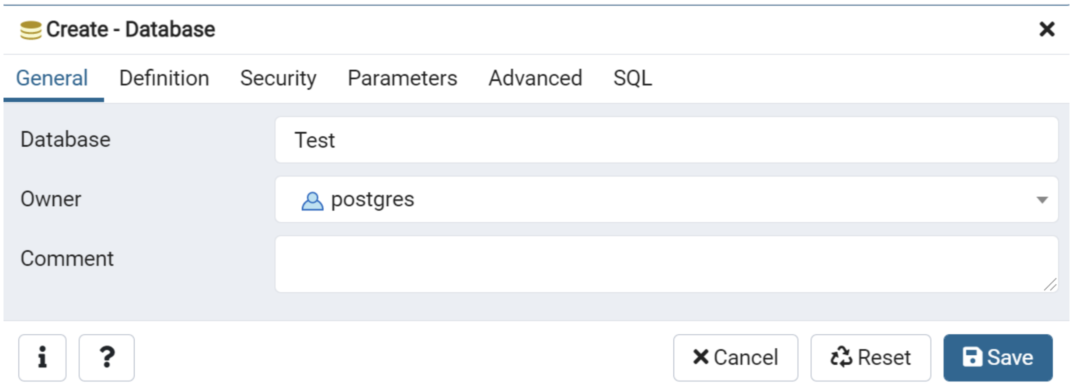
Name	New_Table
Owner	postgres
Schema	production
Tablespace	Select an item...
Partitioned table?	No
Comment	

Figure 20: Table properties view

New Database

To create a new database, perform the following steps:

1. Right-click on **Databases**.
2. Select **Create** and choose **Database**. A **Create-Database** window gets displayed.
3. Enter a database name beside **Database**.
4. Then press the **Save** button:



Database: Test
Owner: postgres
Comment:

Figure 21: Create - Database window with a database name to be created

Your new database is ready.

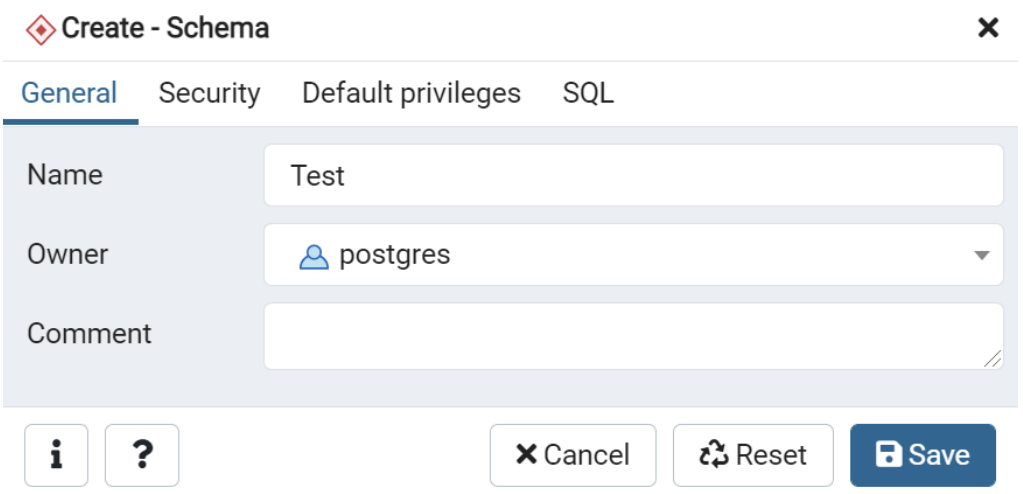
You have grasped how to create a new database. Now you can proceed to create a schema.

New Schema

To create a schema, perform the following steps:

1. Right-click on **Schemas**.
2. Click **Create** and select **Schema**. The **Create – Schema** window gets displayed (*Figure 22*):

3. Enter a name of a schema beside **`Name`** and click the **`Save`** button.



Name: Test

Owner: postgres

Comment:

Figure 22: Create - Schema window for creating a new schema

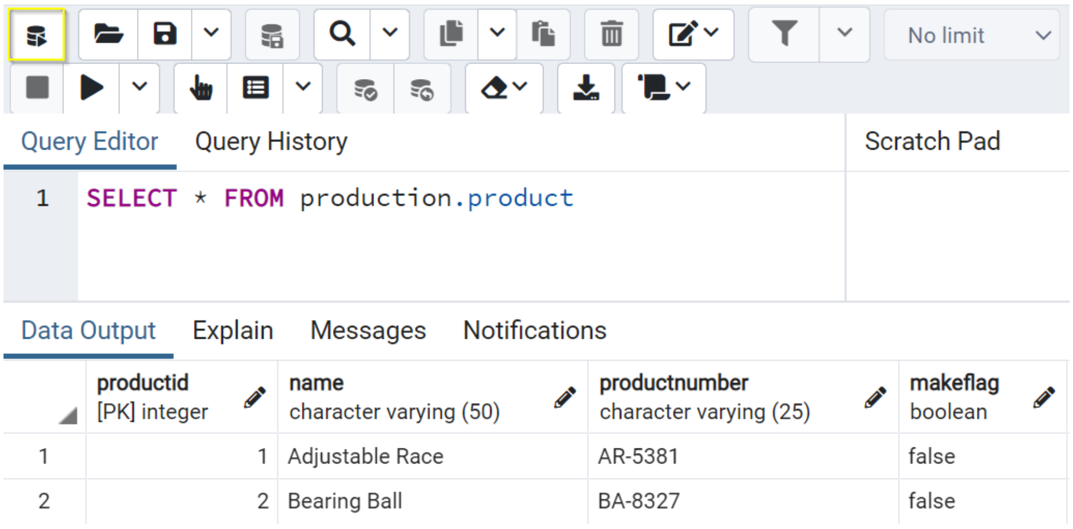
A new schema gets added as a branch under **`Schemas`** on the left side.

This section covered what you can do with the data in a table and how a database can be created. With this knowledge, you can now figure out how you can query and modify data using SQL, as discussed in the following section.

SQL Queries and Commands

To learn how you can query and modify data using SQL, perform the following steps:

1. First right click on the new **`Adventureworks`** database.
2. Then select the pgAdmin **`Query Tool`**. The Query Tool is a powerful tool that allows the execution of arbitrary SQL commands and helps in reviewing the results (*Figure 23*):



```
SELECT * FROM production.product
```

	productid [PK] integer	name character varying (50)	productnumber character varying (25)	makeflag boolean
1	1	Adjustable Race	AR-5381	false
2	2	Bearing Ball	BA-8327	false

Figure 23: Query tool button displaying results view in the Data Output tab

Observe that the top panel displays the SQL Editor, while the lower panel displays the Data Output panel.

You have learned how to query a database using SQL commands. The next step is for you to proceed to know about writing comments in SQL using various SQL commands—**`SELECT`**, **`WHERE`**, **`JOIN`**, **`INSERT`**, **`DELETE`**, **`UPDATE`**, **`ORDER BY`**, **`GROUP BY`**—and Aggregate functions.

COMMENT

Like all programming languages, you can write comments in SQL as well. This is done using two dashes (**`--`**). You can have multi-line comments too (done in the same way as in C#)—that is, using slashes with an asterisk (**`/*random comment*/`**).

SELECT

The task you will perform most often is querying a SQL database. SQL reads like very simple English, so even if you have no experience with it, you can easily intuit what is happening just by looking at it.

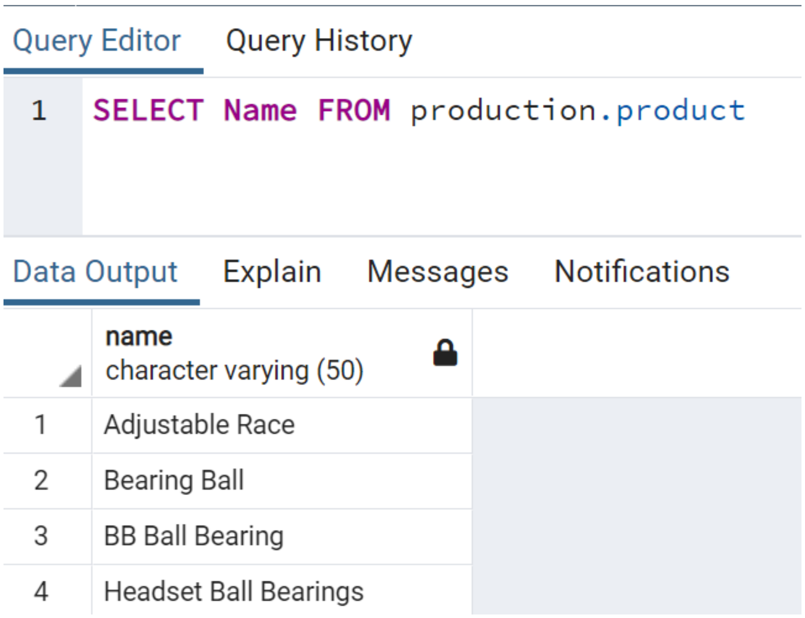
SELECT is the statement used to get data from a table. This data can be from one or more columns. The **SELECT** statement syntax could be defined like this:

```
SELECT column1, column2, ...
FROM table
```

For example, consider the following query:

```
SELECT Name FROM production.product
```

This will retrieve the names of all the products:



```
SELECT Name FROM production.product
```

```
  name
  character varying (50)
1 Adjustable Race
2 Bearing Ball
3 BB Ball Bearing
4 Headset Ball Bearings
```

Figure 24: The query in the Query Editor window showing results in the Data Output window

It's important to note that the casing of a query matters, especially in PostgreSQL. That is because PostgreSQL often converts everything to lowercase, and therefore most database objects will be named in lowercase to work smoothly with the database engine. So, when using this database, try to use lowercase characters exclusively. If you want to separate words, do so using **snake_case**, where new words are separated by an underscore (_).

If you have paid close attention, you might have noticed that you kind of violated the rule of all-lowercase when you wrote your first select, writing **`Name`**. If you define table or column names in all-lowercase, you can still write queries to work with them using mixed capitalization (like you did just now). However, if an object name has mixed capitalization, you will need to add quotes. For example, if the **`Product`** table belonged to a schema named **`Production`** (note the capitalization of the letter **`P`**), to escape PostgreSQL capitalization problems, you would need to add double quotes on each while querying, as follows:

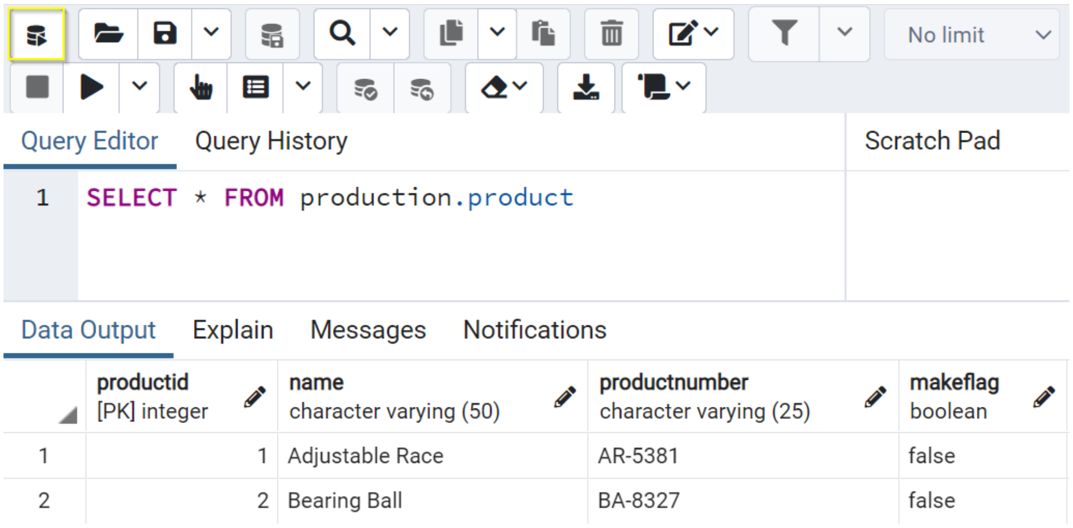
```
SELECT name FROM "Production"."Product"
```

The general convention in PostgreSQL is to capitalize keywords and use non-capital letters for tables, columns, or other objects.

Sometimes, you simply want to select everything that a table has. In that case, you can use a **wildcard** symbol (**`*`**). For example, consider the following query:

```
SELECT * FROM production.product
```

You should see the following output on executing this query:



```
SELECT * FROM production.product
```

	productid [PK] integer	name character varying (50)	productnumber character varying (25)	makeflag boolean
1	1	Adjustable Race	AR-5381	false
2	2	Bearing Ball	BA-8327	false

Figure 25: Query for selecting all data from the production.product table

To inspect the content of a database, using a wildcard in a **`SELECT`** statement is fine. However, it's not recommended for production. You won't usually want to get all data (because it's more expensive as it's a bigger payload), so you should explicitly specify what you want to query.

WHERE

As described earlier, you don't typically want to retrieve all the data in a database, but rather data about a specific object. The **`WHERE`** statement allows you to filter data and return it based on some condition. The syntax of **`WHERE`** is very similar to the syntax of **`if`**, as it also returns values based on a predicate. In the following example, you need information about an item by using the **`WHERE`** statement to get that item based on its **`productid`** value:

```
SELECT * FROM production.product
WHERE productid = 1
```

This query selects a product for which the value of **`productid`** is **`1`**.

JOIN

A very common operation when working with databases is combining data from two or more tables. There are multiple kinds of joins; however, here the most important ones will be talked about: left joins and inner joins. They are the most important because using them, you can achieve what all the other joins achieve.

INNER JOIN is an operation used for getting two linked tables. It returns data only when a link between two columns satisfies a condition. You can use **`INNER JOIN`** when you want to return only data that must exist in both tables based on a condition. The syntax for this is as follows:

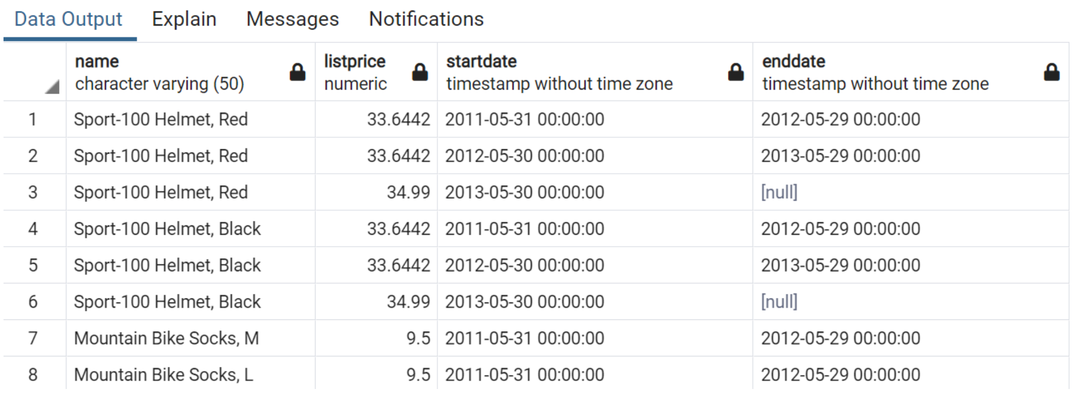
```
SELECT table1.column1, table2.column2, ...
FROM table1
INNER JOIN table2
ON table1.foreignKey = table2.primaryKey
```

This query takes columns from both **`table1`** and **`table2`**. It can determine which rows from these two tables should be joined by the value of **`table1.foreignKey`** matching **`table2.primaryKey`**.

For example, consider the following:

```
SELECT production.product.name,
production.productlistpricehistory.listprice,
production.productlistpricehistory.startdate,
production.productlistpricehistory.enddate
FROM production.product
INNER JOIN production.productlistpricehistory
ON production.product.productId = production.productlistpricehistory.
productid
```

The preceding query gets every product's name and details regarding the changes in their prices over time by joining the **`product`** and **`productlistpricehistory`** tables, based on the matching **`productid`** value:



	name character varying (50)	listprice numeric	startdate timestamp without time zone	enddate timestamp without time zone
1	Sport-100 Helmet, Red	33.6442	2011-05-31 00:00:00	2012-05-29 00:00:00
2	Sport-100 Helmet, Red	33.6442	2012-05-30 00:00:00	2013-05-29 00:00:00
3	Sport-100 Helmet, Red	34.99	2013-05-30 00:00:00	[null]
4	Sport-100 Helmet, Black	33.6442	2011-05-31 00:00:00	2012-05-29 00:00:00
5	Sport-100 Helmet, Black	33.6442	2012-05-30 00:00:00	2013-05-29 00:00:00
6	Sport-100 Helmet, Black	34.99	2013-05-30 00:00:00	[null]
7	Mountain Bike Socks, M	9.5	2011-05-31 00:00:00	2012-05-29 00:00:00
8	Mountain Bike Socks, L	9.5	2011-05-31 00:00:00	2012-05-29 00:00:00

Figure 26: Product names retrieved depending on price changes over time

LEFT JOIN is another kind of join used for getting two linked tables. The key difference compared to **`INNER JOIN`** is that, even if the predicate condition is not satisfied, data will still be returned, just with null values of the other table. You should use it when you are working with optional data and just want to append data from the table you are joining to. In other words, data doesn't have to exist on the right table.

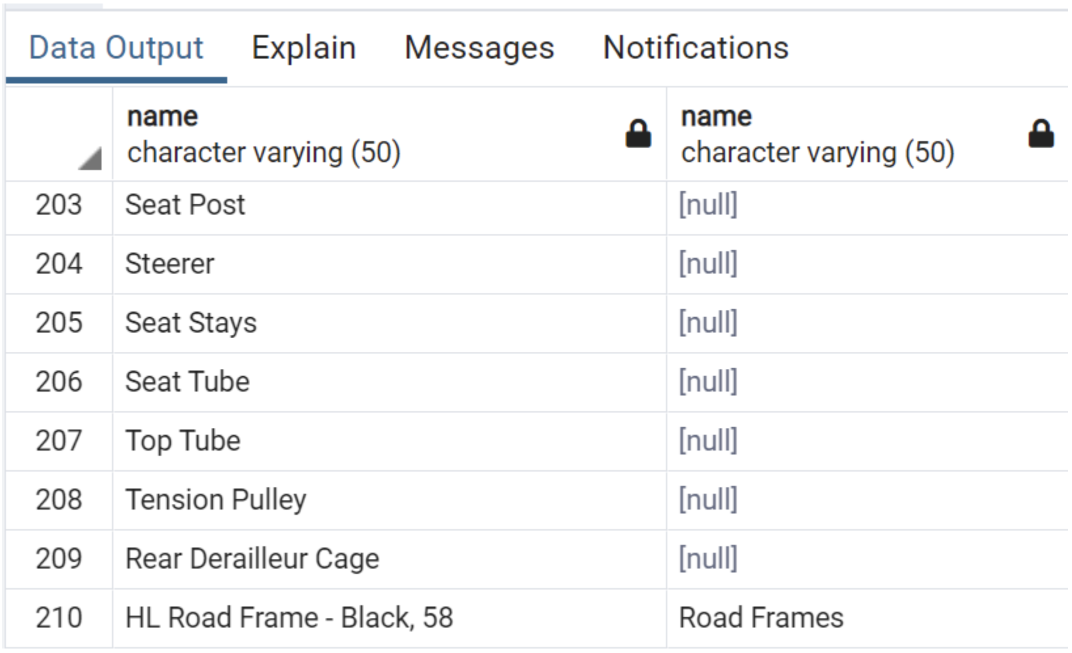
The syntax is nearly identical to the previous query:

```
SELECT table1.column1, table2.column2, ...
FROM table1
LEFT JOIN table2 ON table1.column3 = table2.column4
```

For example, consider the following:

```
SELECT production.product.name, production.productsubcategory.name
FROM production.product
LEFT JOIN production.productsubcategory
ON production.product.productsubcategoryid = production.
productsubcategory.productsubcategoryid
```

The preceding query gets every product name and its subcategory (or **`[null]`** for subcategory if a product has no subcategory), as shown in *Figure 27*:



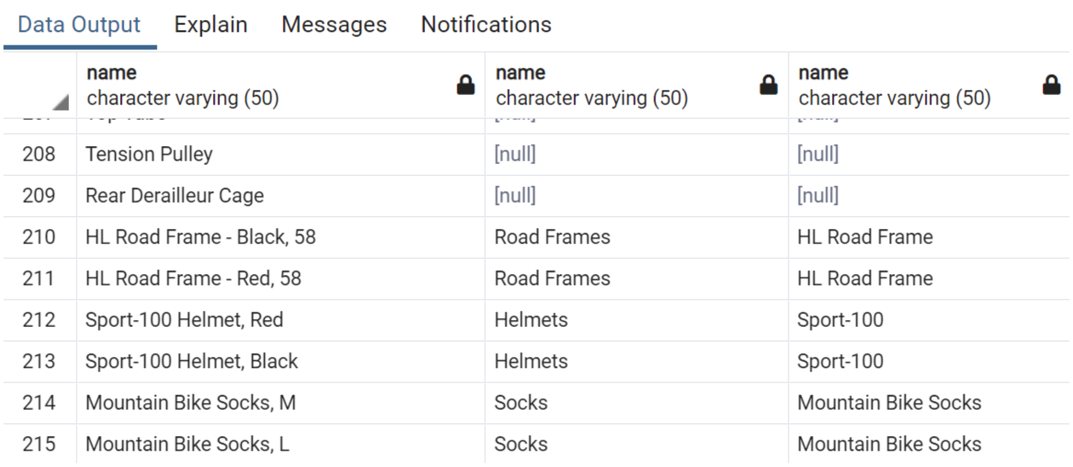
	name character varying (50)	name character varying (50)
203	Seat Post	[null]
204	Steerer	[null]
205	Seat Stays	[null]
206	Seat Tube	[null]
207	Top Tube	[null]
208	Tension Pulley	[null]
209	Rear Derailleur Cage	[null]
210	HL Road Frame - Black, 58	Road Frames

Figure 27: Product and its subcategory names

You can join more than one table, but the syntax remains much the same. For example, if you also want to get the **`name`** of a **`productmodel`**, the following syntax could be written:

```
SELECT production.product.name, production.productsubcategory.name,
production.productmodel.name
FROM production.product
LEFT JOIN production.productsubcategory
ON production.product.productsubcategoryid = production.
productsubcategory.productsubcategoryid
LEFT JOIN production.productmodel
ON production.product.productmodelid = production.productmodel.
productmodelid
```

The preceding query looks for products (their **`name`**) and their matching subcategories (their **`name`**) and matching models (their **`name`**). If a product doesn't have a model or a subcategory, the column will have a null value. You'll see the following output upon executing this query:



	name character varying (50)	name character varying (50)	name character varying (50)
208	Tension Pulley	[null]	[null]
209	Rear Derailleur Cage	[null]	[null]
210	HL Road Frame - Black, 58	Road Frames	HL Road Frame
211	HL Road Frame - Red, 58	Road Frames	HL Road Frame
212	Sport-100 Helmet, Red	Helmets	Sport-100
213	Sport-100 Helmet, Black	Helmets	Sport-100
214	Mountain Bike Socks, M	Socks	Mountain Bike Socks
215	Mountain Bike Socks, L	Socks	Mountain Bike Socks

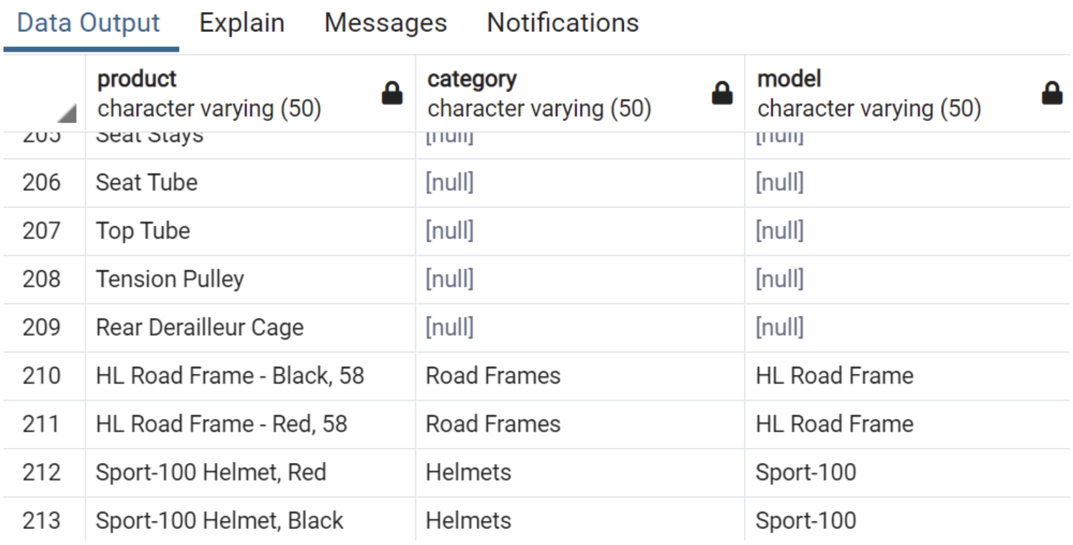
Figure 28: Product, product subcategory, and model names displayed

The query returns all the products and their subcategory and model names but will do so only if such names exist. If you read the results table, you will notice an odd thing; the three returned columns are **`name`**, **`name`**, and **`name`**. That's not very descriptive.

This problem can be solved by giving aliases to the results returned. **Aliases** can be provided by using the **AS** keyword. All you must do is add **`as othername`** after a column in a **`SELECT`** statement, where othername refers to an alias of your choice. Aliases can be given to tables as well. Using aliases, the query written earlier could be simplified as follows:

```
SELECT product.name as product, category.name as category, model.name as
model
FROM production.product as product
LEFT JOIN production.productsubcategory as category
ON product.productsubcategoryid = category.productsubcategoryid
LEFT JOIN production.productmodel as model
ON product.productmodelid = model.productmodelid
```

The previous result ignored table names when presenting the results in a single view, therefore you saw 3 **`name`** columns. The preceding query replaces **`product.name`**, **`category.name`** and **`model.name`** column names with aliases. Instead of the **`name`**, **`name`**, and **`name`** columns, you have **`product`**, **`category`**, and **`model`**, which are far more descriptive:



	product character varying (50)	category character varying (50)	model character varying (50)
205	Seat Stays	[null]	[null]
206	Seat Tube	[null]	[null]
207	Top Tube	[null]	[null]
208	Tension Pulley	[null]	[null]
209	Rear Derailleur Cage	[null]	[null]
210	HL Road Frame - Black, 58	Road Frames	HL Road Frame
211	HL Road Frame - Red, 58	Road Frames	HL Road Frame
212	Sport-100 Helmet, Red	Helmets	Sport-100
213	Sport-100 Helmet, Black	Helmets	Sport-100

Figure 29: Results of the query with three ambiguous name columns aliased as product, category, and model

INSERT

Adding a new row in SQL is done with an **`INSERT`** command. The syntax is as follows:

```
INSERT INTO table1
(column1, column2, ...)
VALUES (value of column 1, value of column 2)
```

The order in which you specify columns does not matter. However, the values that follow need to be in the same order as the columns specified. For example, to create a new product with some basic data, you could run the following query:

```
INSERT INTO production.product
(productid, name, productnumber, standardcost, listprice, sellstartdate,
safetystocklevel, reorderpoint, daystomanufacture)
VALUES (100001, 'New Product', 9999, 15.7, 999, '2021-05-25', 1, 1, 2);
```

This query can be read as follows: insert a new row in the **`production.product`** table, and for the **`productid`**, **`name`**, **`productnumber`**, **`standardcost`**, **`listprice`**, **`sellstartdata`**, **`safetystocklevel`**, **`reorderpoint`**, and **`daystomanufacture`** columns, assign the values **`100001`**, **`New Product`**, **`9999`**, **`15.7`**, **`999`**, **`2021-05-25`**, **`1`**, **`1`**, and **`2`**, respectively.

DELETE

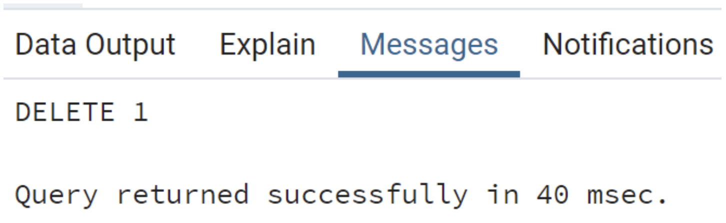
Removing data is quite simple; however, that's also why you should be careful. A reckless **`DELETE`** statement could remove all data by executing a command of just a few words. The syntax is as follows:

```
DELETE FROM table
WHERE condition
```

For example, consider the following:

```
DELETE FROM production.product
WHERE productid = 100001
```

This query deletes the item with **`productid = 100001`**. Note that executing SQL commands that modify a table will also show how many rows were affected:



```
DELETE 1

Query returned successfully in 40 msec.
```

Figure 30: Message showing a successful delete on the affected rows

You can execute a **`DELETE`** command without a condition. However, as you may have guessed already, this would result in all data in that table being lost. For example, if you removed the **`WHERE productid = 100001`** clause, you would suddenly lose all data from the **`production.product`** table. Therefore, it's always efficient to pair a **`DELETE`** statement with a **`SELECT`** statement—that is, to first select the data that you want to delete and only then perform the actual deletion. An example of this is shown in the following snippet:

```
SELECT * FROM production.product
WHERE weight < 1000 AND weight > 50

DELETE FROM production.product
WHERE weight < 1000 AND weight > 50
```

This query will first select all products with weights of less than **`1000`** and more than **`50`**. Then, if you get the results you expected, you can perform a delete with the same condition.

UPDATE

The last data modification command is **`UPDATE`**. The syntax for this is as follows:

```
UPDATE table
set column1 = value1,
column2 = value2
WHERE condition
```

For example, see the following query:

```
UPDATE production.product
SET color = 'Green'
WHERE color = 'Red'
```

This query changes the value of **`color`** from **`Red`** to **`Green`**, wherever applicable. This command carries a similar risk as **`DELETE`** because, if no condition is provided, you may end up updating all table rows to have the same values. Again, for that reason, it's efficient to pair it with a **`SELECT`** statement.

ORDER BY

Often, you will need to sort data in a column in a certain order. This can be done using an **`ORDER BY`** statement. The syntax for this is as follows:

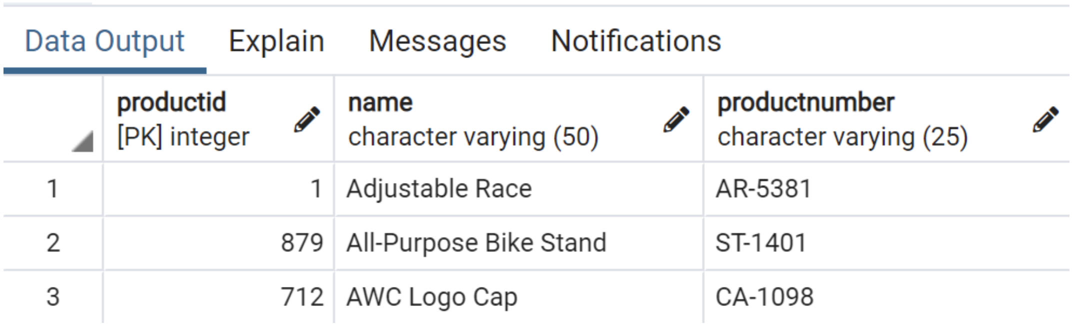
```
SELECT column1, column2...

FROM table
ORDER BY column1
```

For example, consider the following query:

```
SELECT *
FROM production.product
ORDER BY name
```

This query retrieves data from all the products and then sorts them alphabetically by product name (*Figure 31*):



	productid [PK] integer	name character varying (50)	productnumber character varying (25)
1	1	Adjustable Race	AR-5381
2	879	All-Purpose Bike Stand	ST-1401
3	712	AWC Logo Cap	CA-1098

Figure 31: Products arranged alphabetically by their name

You can choose the order of sorting using the **ASC** or **DESC** keywords. By default, the sort order is ascending. For example, see the following query:

```
SELECT name, listprice
FROM production.product
ORDER BY listprice DESC
```

This gets all products' names and their prices, sorted in descending order.

GROUP BY AND AGGREGATE FUNCTIONS

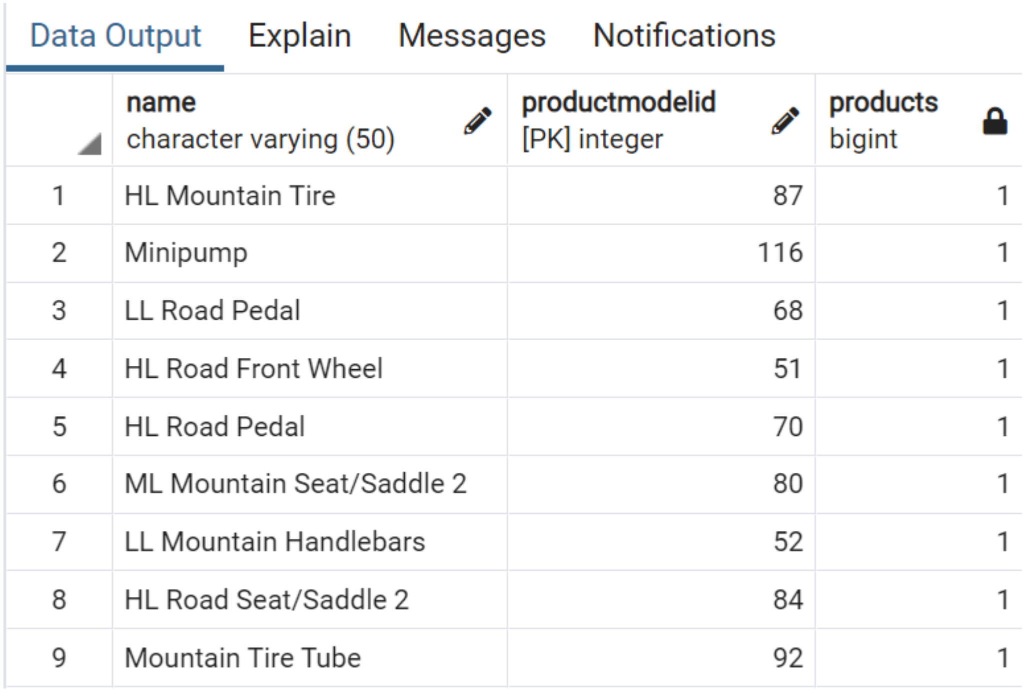
Some statements don't work on data as is. Sometimes, data needs to be prepared in advance. Such statements are called **aggregate functions**. Examples of these include **COUNT**, **MAX**, **MIN**, and **SUM**.

When dealing with aggregate functions, you first need to use a **GROUP BY** statement to logically group data, as done in the following query:

```
SELECT model.name, count(model.name) as products
FROM production.product product
INNER JOIN production.productmodel model
ON product.productmodelid = model.productmodelid
GROUP BY model.name
```

The preceding query returns the total number of products that each model has. In this query, every product with the same name is put into a group. When using the **GROUP BY** statement, you are working with groups of data and can therefore perform operations specific to groups.

In this case, you are counting the number of products in the groups. You got all products for every model and grouped them by unique model names. The result is displayed in *Figure 32*:



Data Output Explain Messages Notifications

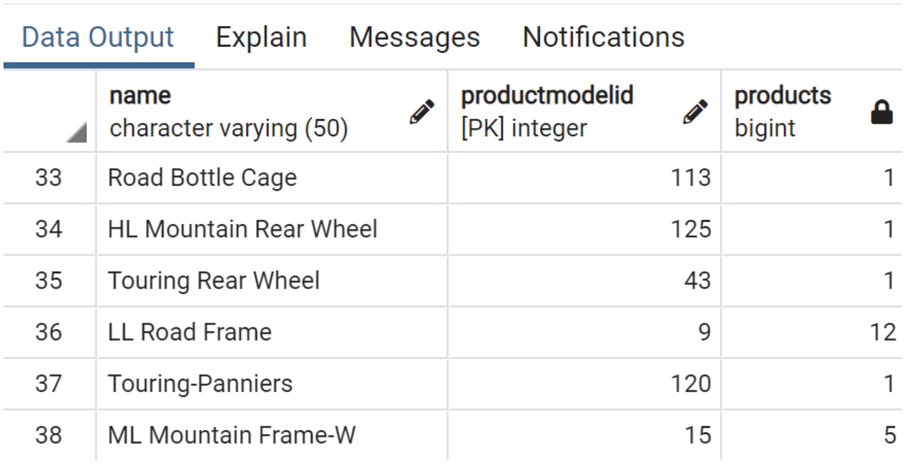
	name character varying (50)	productmodelid [PK] integer	products bigint
1	HL Mountain Tire	87	1
2	Minipump	116	1
3	LL Road Pedal	68	1
4	HL Road Front Wheel	51	1
5	HL Road Pedal	70	1
6	ML Mountain Seat/Saddle 2	80	1
7	LL Mountain Handlebars	52	1
8	HL Road Seat/Saddle 2	84	1
9	Mountain Tire Tube	92	1

Figure 32: Unique product model names with their products counts

The important thing to note about aggregate functions is that the selected columns also need to be grouped. So, to select **`productmodelid`** along with **`name`** and count the number of **`products`** that exist for each model, you need to put both columns into a **`GROUP BY`** statement, as done in the following query:

```
SELECT model.name, model.productmodelid, count(model.name) as products
FROM production.product product
INNER JOIN production.productmodel model
ON product.productmodelid = model.productmodelid
GROUP BY model.name, model.productmodelid
```

Grouping by `productmodelid` does not affect other columns because there will be a unique `productmodelid` value but only across different groups. Therefore, it is safe to select it (and not mix it with a random element from that group). *Figure 33* shows the result of running the preceding query:



	name character varying (50)	productmodelid [PK] integer	products bigint
33	Road Bottle Cage	113	1
34	HL Mountain Rear Wheel	125	1
35	Touring Rear Wheel	43	1
36	LL Road Frame	9	12
37	Touring-Panniers	120	1
38	ML Mountain Frame-W	15	5

Figure 33: The productmodelid selected on top of the product counts of different models

Don't repeat yourself; this principle applies to SQL as much as most other languages. If you find yourself writing the same queries all over again, it might be effective to putting them in a function. With SQL, you have two options: **functions** and **stored procedures**. Writing and using them requires quite a lot of focus on a database, and as the focus for this is on C# and Entity Framework, it is better to skip going deeper into the topic.

NOTE

If you do want to familiarize yourself more with functions and stored procedures, refer to https://www.postgresql.org/docs/11/sql-createprocedure.html.

This section covered the SQL queries and commands that are used to query and modify data using SQL. With this knowledge, you can now proceed with relational database design theory as discussed in the following section.

Relational Database Design Theory—Normalization

What does an effective database look like? This question is answered with a very specific theory—data normalization. **Data normalization** is a theory that tries to define the best practices of how to design tables and relationships between them so that data is unique across rows and keeps its integrity. A normalization form aims to improve database design so that there is as little data duplication as possible. Every form is a set of extra rules, where the rules from previous forms also apply. There are six **normalization forms** (**NFs**) in total; however, in practice, only the first three forms matter in most cases, so those will be the focus here.

1NF

In SQL, as you know already, you should not waste space in tables. Therefore, all rows should be unique. On top of that, every column should hold only **one value**. A violation of this form would be a full name column when you want to access the first name and surname separately. If, however, you always want to access the two through a combination, **`fullname`**, then 1NF would not be violated. Generally, you should be careful about combining too much information into a single column and splitting it instead. Columns should contain atomic (logically single) values.

2NF

Every column in a row should be related only to **key column(s)**—either a single column key or a composite key. Consider an example of a table, **`dogowner`**, that contains columns named **`personid`**, **`personname`**, **`person age`**, **`dogname`**, and **`dogage`**. A person can own a dog, but the dog's name and age are not directly related to the person. Therefore, the table should be split into two: **`dog`** (with columns for **`id`**, **`name`**, and **`age`**) and **`person`** (with columns for **`id`**, **`name`**, and **`age`**). Though semantically the same, the two tables are logically different. Each one is likely to change in the future, and so the changes in one would not impact the other.

3NF

Just like 2NF, in 3NF all columns should uniquely talk about the primary key column(s). The key difference between 2NF and 3NF is the word **transitive**, meaning that you should not have columns that could be deducted based on a non-key column.

For example, suppose you have a table named **person**, with columns for **id**, **name**, **age**, **bmi**, and **bodystatus**. All these factors uniquely describe a single person. However, **bodystatus** depends on **bmi**. For example, **bmi** values under 18.5 mean that the person is underweight, while values between 18.5 and 24.9 mean that the person's weight is in the normal range.

The problem with having **bodystatus** in the same table as **bmi** is that many people can have the same **bodystatus** value. What if the **bmi** metrics for **bodystatus** change? That will be problematic because multiple rows need to be updated. One solution to this problem would be that instead of **hardcoding** **bmi** values, you could split the table into two: a **person** table with their body metrics, and a **bodystatus** table with **statusname**, **minbmi**, and **maxbmi** columns (to indicate a range of BMI for that status). Another way would be to use a SQL function to return a **bodystatus** value based on the **bmi** value.

> **NOTE**
>
> To read more about data normalization basics, refer to the Microsoft docs at https://docs.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description.

RELATIONS

In relational database design, there are three kinds of relations: 1:n, 1:1, and n:n. All of them are related to primary and foreign key relations. Grasping how these relationships work is key to an effective database design.

1:N

1:n reads as **one-to-many relation** and is the most common relation wherein one table refers to another through a foreign key, and that reference is one-sided. The best way to describe all the relationships is through **JOIN** statements.

A **JOIN** statement for 1:n looks like this:

```
SELECT *
FROM table1
INNER JOIN table2
ON table1.table2Id = table2.Id
```

For example, a product has a model associated with it, but the same model can be applied to multiple products. This means that a model has many products. Another example is a warehouse. A specific item belongs to a specific warehouse, but there are many items in a warehouse. Thus, an item and a warehouse have a one-to-many relationship.

1:n is the most common relationship, and it's usually practical to stick to it. This is because it directly links to all the related data without any problems. (This idea will become clearer in the next couple of sections.)

1:1

1:1 reads as **one-to-one relation** and is a relation in which parts of the same entity are split across two tables. Sometimes, this relation is defined through a join of primary keys of two tables, such as the following:

```
SELECT *
FROM table1
INNER JOIN table2
ON table1.Id = table2.Id
```

Since both tables are uniquely identified, matching their ID columns (given there is a match) will result in only one match, one row. This kind of relation is often used when there is extra information about the main table. Often, that extra information will be optional.

For example, assume that there is a conference where information on guests is stored in a database. A speaker can also be a guest. So, the guest and speaker make a 1:1 relation. This relation is especially useful here because guests will have a single empty column (FK to speaker table) if they are not a speaker.

A 1:1 relation does not only mean that a table currently has a **single link**; it also means that it may never have more links (logically). The choice of having such a relationship is usually purely for performance details (not including or storing null values for data that is likely to be optional). However, the difference between 1:1 and 1:n is small and usually, such a relationship is removed by combining two tables into one.

N:N

n:n reads as **many-to-many relation** and is a relation when two tables refer to each other. It's the only relation that is strictly forbidden because it causes problems when modifying data (instead of data in one place being changed, multiple places need to be changed due to the double link).

Consider this example: a person plays many games, and a game has many players. This is a problematic relation because the tables will look like this:

```
person: id, name, gameid
game: id, name, personid
```

And so, every row in the **`person`** table will look like this:

```
1, Tom, 1
1, Tom, 2
2, Tim, 1
...
```

The problem here is that you duplicate the data. The solution, like always, is to split the tables. The n:n relations should always be replaced by two 1:n relations. This is done by putting a table in between, with both primary keys serving as a composite key to identify a unique combination of a player game. Therefore, the tables will now look like this:

```
person: id, name
game: id, name
gameperson: personid, gameid
```

Where are the two 1:n relations? **`Person.id`** makes a foreign key link with the ID of the **`person`** table and **`game.id`** makes a foreign key link with the ID of the **`game`** table. The following query returns all the games and their players:

```
SELECT person.name, game.name
FROM dbo.gameperson as gp
INNER JOIN dbo.person as person
ON gp.personid = person.id
INNER JOIN dbo.game as game
ON gp.gameid = game.idNote
```

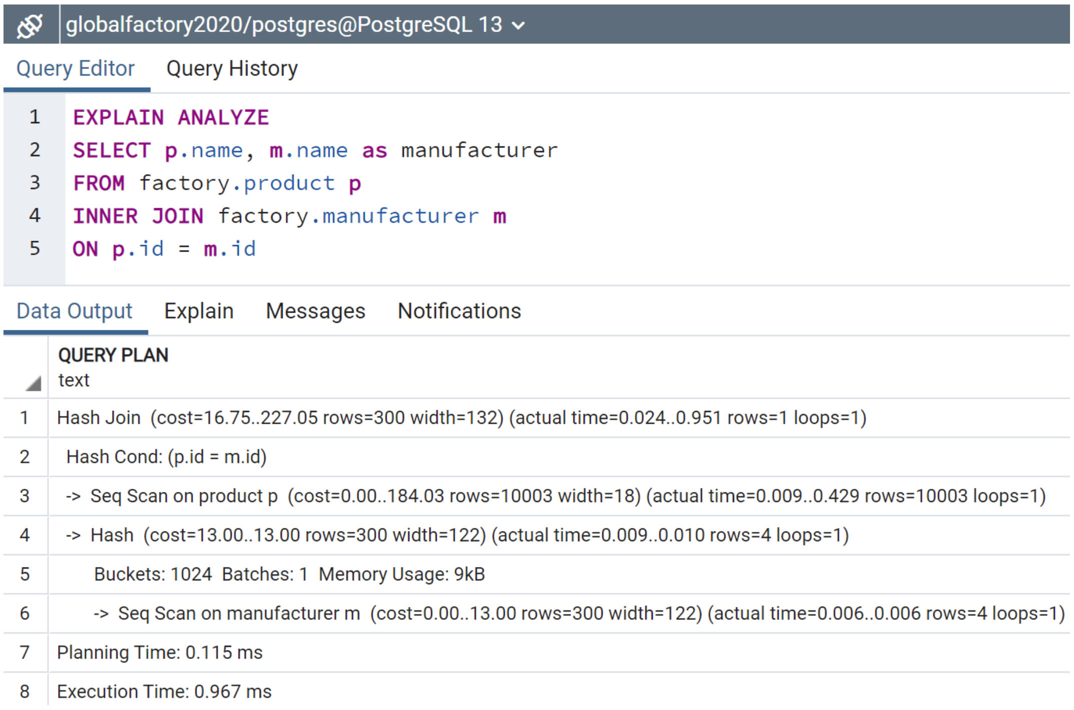
Here **`dbo`** is often a default name for schema, especially in SQL Server databases. If a person does not play any games, nothing will be returned.

NFs and three kinds of relations—1:n, 1:1, and n:n—are the key to a strong database design. Now proceed to learn about SQL query analysis.

QUERY EXECUTION PLAN

Before wrapping this chapter up, it is worth talking about SQL query analysis (in other words, the slow parts of SQL or the detection of bottlenecks). Most databases support functionality to break down every step of a query and produce performance metrics: how fast it ran, how many rows were read, how much data was retrieved, and more. This can be observed in the **query execution plan**.

To have a look at a query execution plan, before the query, write **`EXPLAIN ANALYZE`** as shown in *Figure 34*:



```
EXPLAIN ANALYZE
SELECT p.name, m.name as manufacturer
FROM factory.product p
INNER JOIN factory.manufacturer m
ON p.id = m.id
```

```
QUERY PLAN
text
1  Hash Join  (cost=16.75..227.05 rows=300 width=132) (actual time=0.024..0.951 rows=1 loops=1)
2   Hash Cond: (p.id = m.id)
3   -> Seq Scan on product p  (cost=0.00..184.03 rows=10003 width=18) (actual time=0.009..0.429 rows=10003 loops=1)
4   -> Hash  (cost=13.00..13.00 rows=300 width=122) (actual time=0.009..0.010 rows=4 loops=1)
5        Buckets: 1024  Batches: 1  Memory Usage: 9kB
6        -> Seq Scan on manufacturer m  (cost=0.00..13.00 rows=300 width=122) (actual time=0.006..0.006 rows=4 loops=1)
7  Planning Time: 0.115 ms
8  Execution Time: 0.967 ms
```

Figure 34: Simple join query execution plan

Here, you output an execution plan to get all products and their manufacturers. This is an extremely helpful tool for everyone who wants to not just work with EF, but optimize SQL to the max. Even with a query that ultimately works, there may be room for improvement or greater functionality. A query execution plan is a great tool for finding bottlenecks in queries and eliminating them through optimizations.

By now you are familiar with PostgreSQL and executing commands. So, let's work on all that was learned through the following exercise.

Exercise 1: Creating New Database Factory with Products and Manufacturers

Your local toy factory has hired you as a database developer. They have plans to expand globally. To help with this, they've asked you to create a new database called **`globalfactory2021`**. Within a new **`Factory`** schema, create two new tables: **`product`** and **`manufacturer`**. The **`product`** table should have columns for **`name`**, **`price`**, **`id`**, and **`manufacturerid`**. The **`manufacturer`** table should have columns for **`name`**, **`country`**, and **`id`**.

Perform the following steps to complete this exercise:

1. Create a new database by right-clicking **`Databases`**, selecting **`Create`**, and then **`Database`** (*Figure 35*):

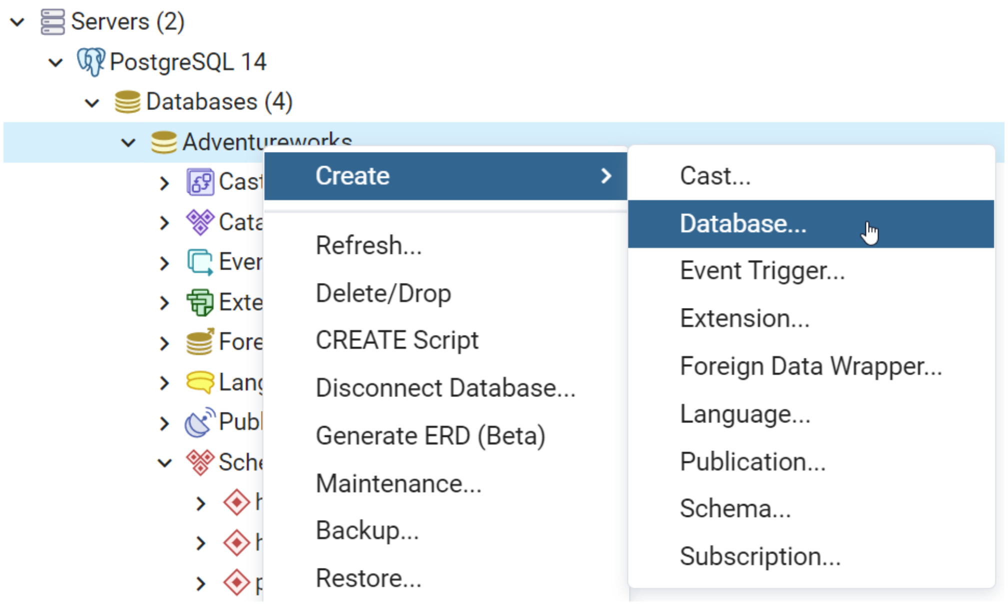
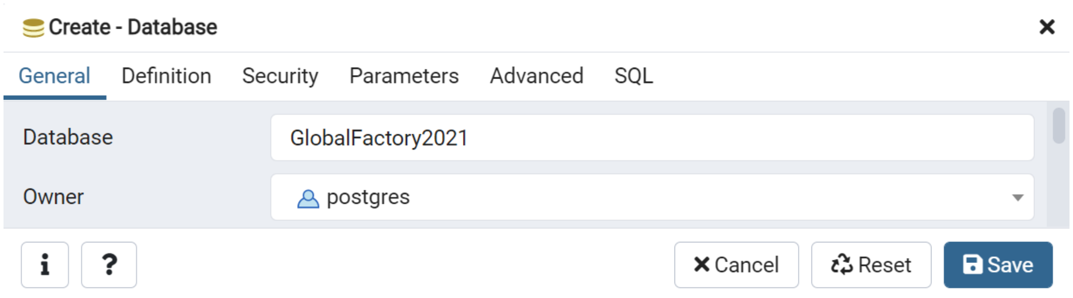


Figure 35: Creating a new database

2. Provide the name of the database as **`GlobalFactory2021`** and then click **`Save`** (*Figure 36*):



Database: GlobalFactory2021
Owner: postgres

Figure 36: Creating the GlobalFactory2021 database

3. Next, create a new schema named **`Factory`**.

4. Expand the **`GlobalFactory2021`** database and then right-click **`Schemas`**, select **`Create`**, and then select **`Schema`**:

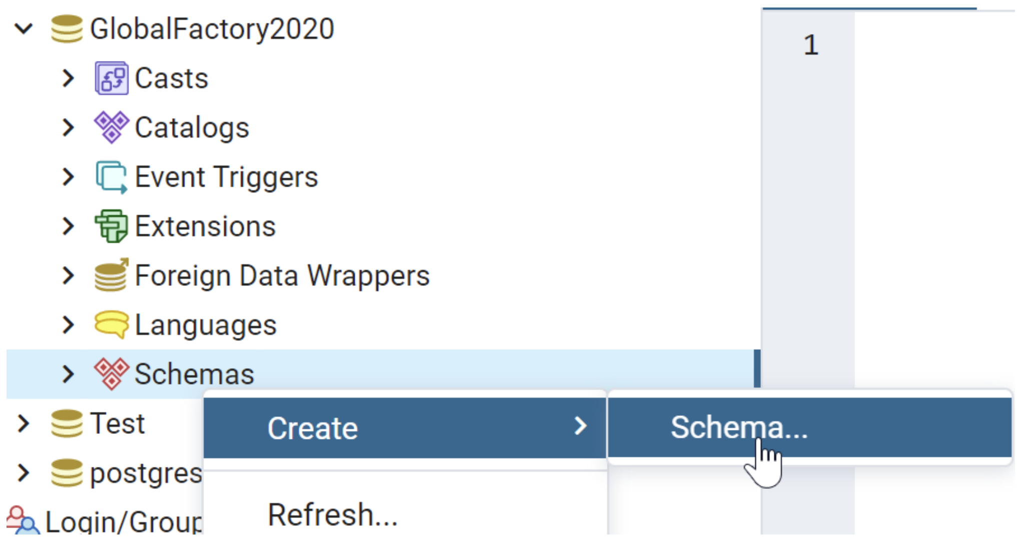
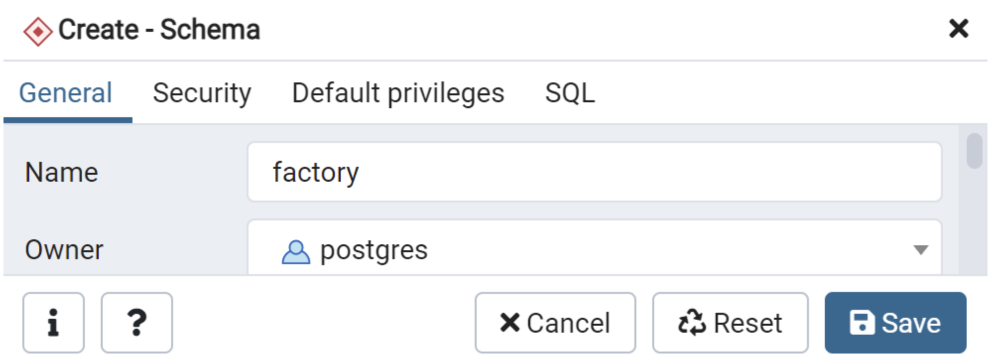


Figure 37: Creating a new schema

5. In the new window, enter **`factory`** and then hit **`Save`** (*Figure 38*):



Name: factory

Owner: postgres

Figure 38: Creating a new Schema called factory

6. To create a new table, expand the factory schema.

7. Right-click on **`Tables`**, select **`Create`**, and then select **`Table`**:

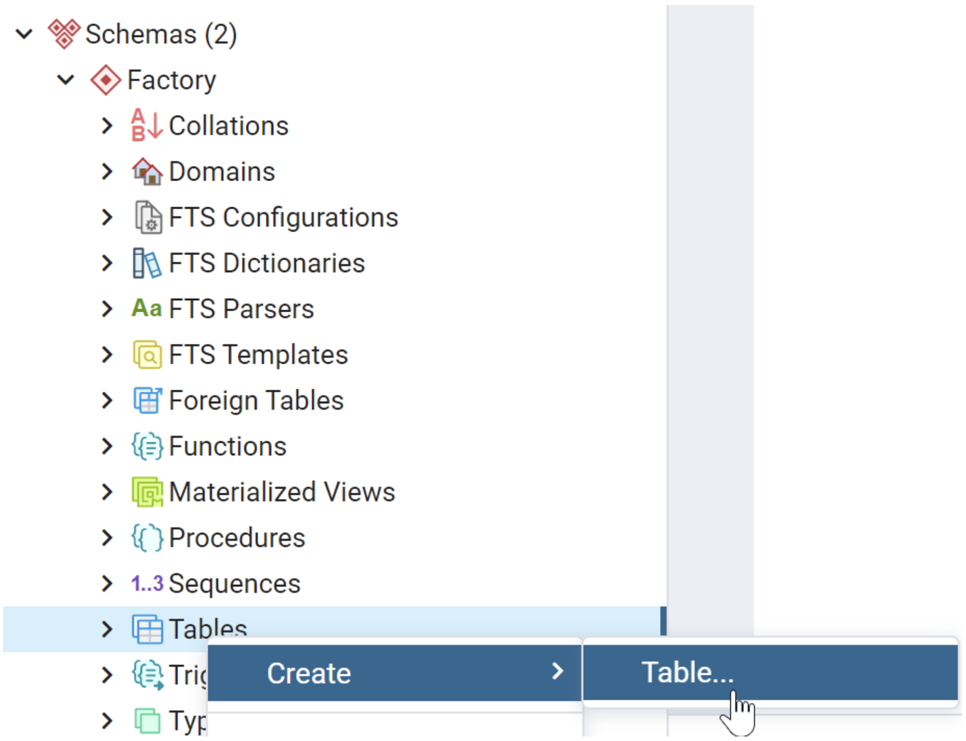
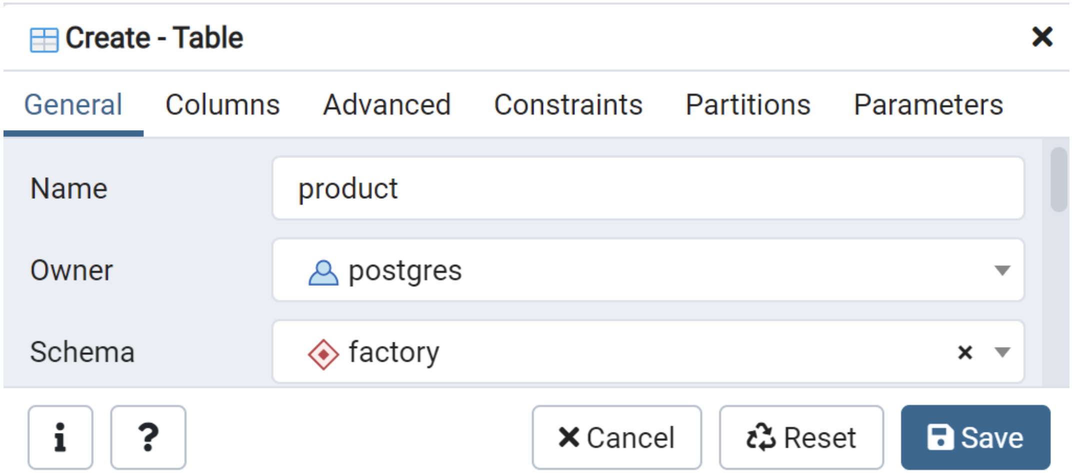


Figure 39: Creating a new table under Schemas

8. Set the table name to **`product`** and then hit **`Save`**:



Name: product

Owner: postgres

Schema: factory

Figure 40: Naming a new table as product

9. Right-click on the **`product`** table and then select **`Properties`** to add columns to the table:

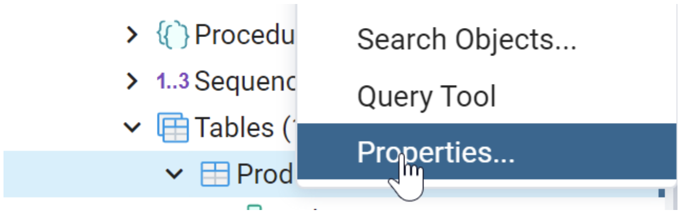
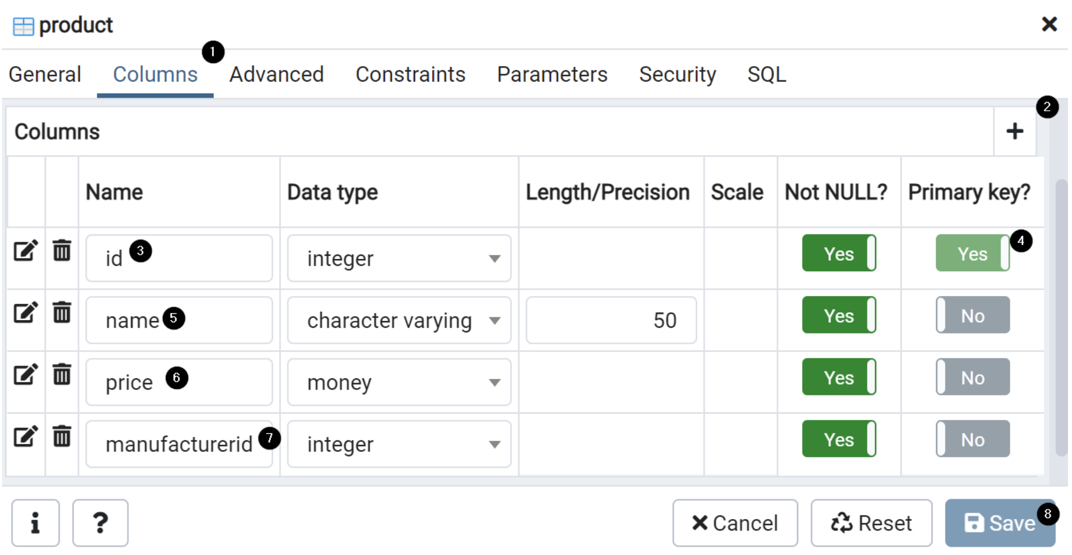


Figure 41: Inspecting table properties

10. Set your **`product`** table columns as shown in *Figure 42*. There are nine sequentially numbered tasks to be followed next.
11. Navigate to the **`Columns`** tab (**1**).
12. Click on **`Add`** (**+**) button (**2**).

13. Type the first column name as **id** (**3**).
14. Set **id** as a **Primary key** (**4**).
15. Set the **Data type** of **id** as **integer** type, **Not Null**.
16. Set the next column **Name** as **name** of **varchar(50)** type, **Not Null** (**5**).
17. Set the **price** of **money** type, **Not Null** (**6**).
18. Set **manufacturerid** of **integer** type, **Not Null** (**7**).
19. Finally, hit the **Save** button (**8**):

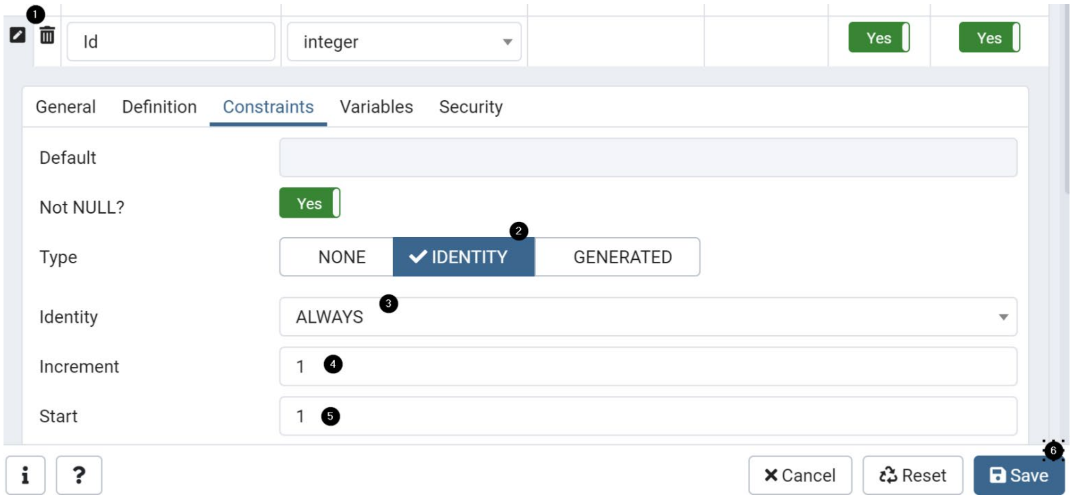


Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?
id	integer			Yes	Yes
name	character varying	50		Yes	No
price	money			Yes	No
manufacturerid	integer			Yes	No

Figure 42: The product table column names and their type set correctly

20. Edit the **id** column as shown in *Figure 43*. There are seven sequentially numbered tasks to be followed next.

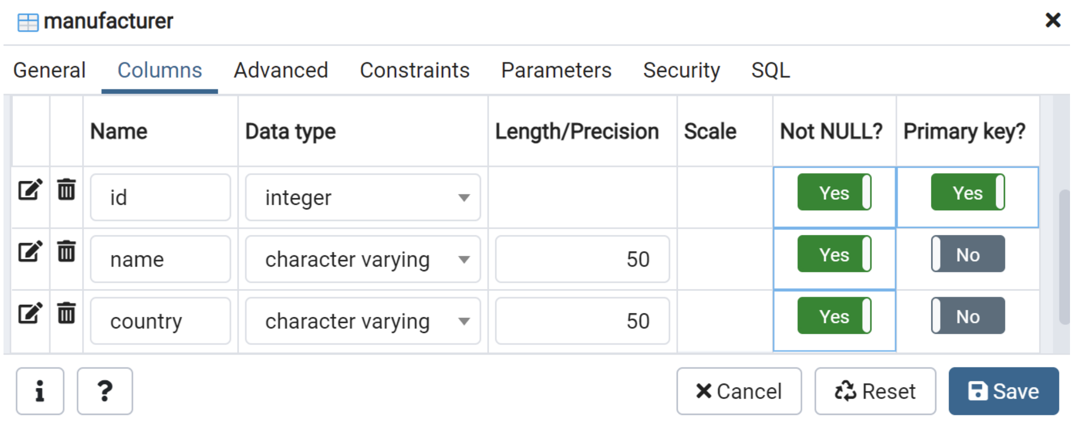
21. Select the **`Edit row`** button beside the **`id`** column (**1**). This is to configure it to make it unique and be incremented by **`1`**.
22. Click the **`Constraints`** tab.
23. Click on **`IDENTITY`** tab (**2**).
24. Choose **`ALWAYS`** from **`Identity`** (**3**).
25. Identity is a group of properties for the id field, which you will often want to start at 1, and for every new row to be incremented by 1. So, mark an **`Increment`** of **`1`** (**4**).
26. Next enter **`1`** beside **`Start`** (**5**).
27. Finally click the **`Save`** button (**6**):



Default	
Not NULL?	Yes
Type	IDENTITY
Identity	ALWAYS
Increment	1
Start	1

Figure 43: Autogenerating Id using 1 as increment

Repeat steps **11** to **19** for the **`factory.manufacturer`** table and create columns named as **`id`**, **`name`**, and **`country`**. Don't forget to seed the primary key, as implemented. The completed table is shown in *Figure 44*:



manufacturer

General | Columns | Advanced | Constraints | Parameters | Security | SQL

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?
id	integer			Yes	Yes
name	character varying	50		Yes	No
country	character varying	50		Yes	No

Figure 44: Columns created to complete factory.manufacturer table

28. As you know, a manufacturer has many products, but a product can have only a single manufacturer. Add this relation by creating a foreign key for **`manufacturerid`** in the **`factory.product`** table (*Figure 45*). There are ten sequentially numbered tasks to be followed next.

29. Click the **`Constraints`** tab (**1**) of the **`product`** table.

30. Choose **`Foreign Key`** tab (**2**).

31. Click on **`Add`** (**+**) button (**3**) to add a new constraint.

32. Then enter the name using the convention **`parenttable_localcolumn_referencingcolumn`**. Here the name entered is **`product_manufacturerid_id`** as per the convention (**4**).

33. Click on the **`Edit row`** button (**5**).

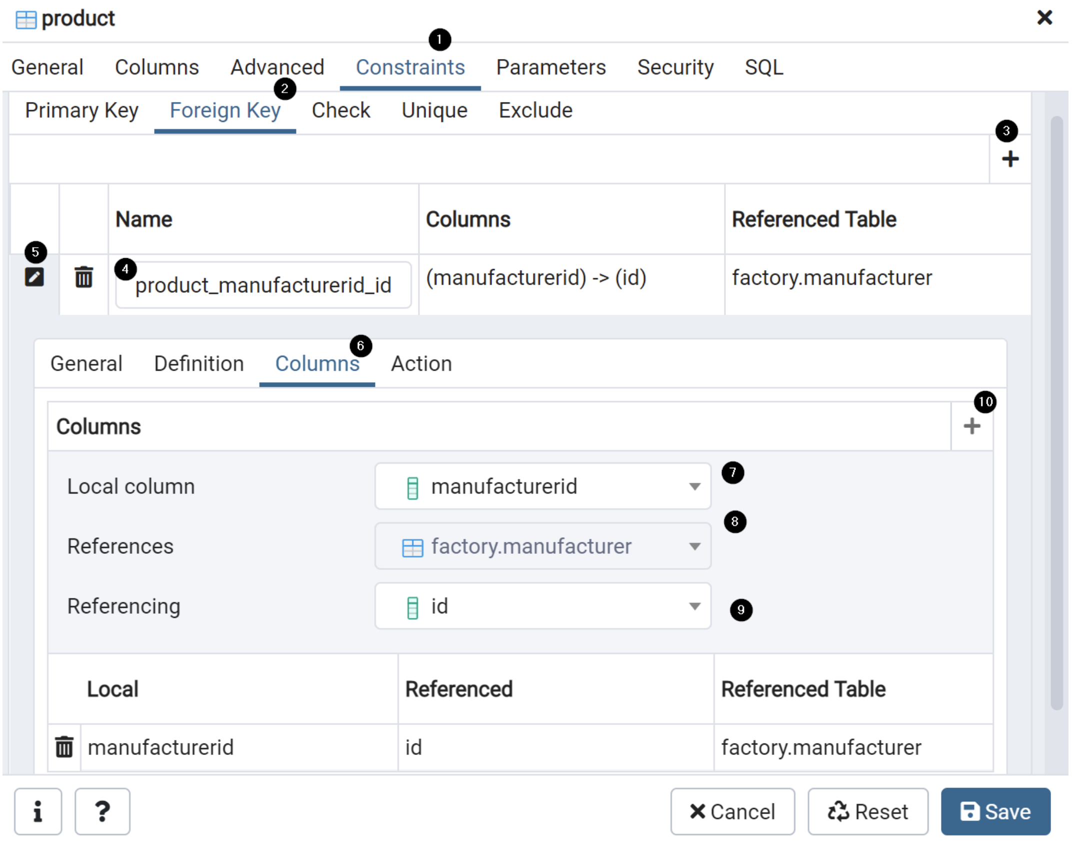
34. Go to **`Columns`** tab (**6**) and enter **`manufacturerid`** beside **`Local column`**, which is the foreign key column (**7**).

35. Beside the **`References`** option, type **`factory.manufacturer`** (**8**). This is needed to specify which table will be linked using a foreign key relation.

36. Type **`id`** beside **`Referencing`** (**9**). This is needed to specify which column in the reference table will be used to make a connection between the two tables.

37. Then, hit the **`Add`** (**+**) (**`10`**) button (top-right corner of **`Columns`** tab). You will see the constraint appear at the bottom of the table.

38. Do **not** press the **`Save`** button or close the window yet:



product

General | Columns | Advanced | Constraints | Parameters | Security | SQL

Primary Key | Foreign Key | Check | Unique | Exclude

Name	Columns	Referenced Table
product_manufacturerid_id	(manufacturerid) -> (id)	factory.manufacturer

General | Definition | Columns | Action

Columns

Local column	manufacturerid
References	factory.manufacturer
Referencing	id

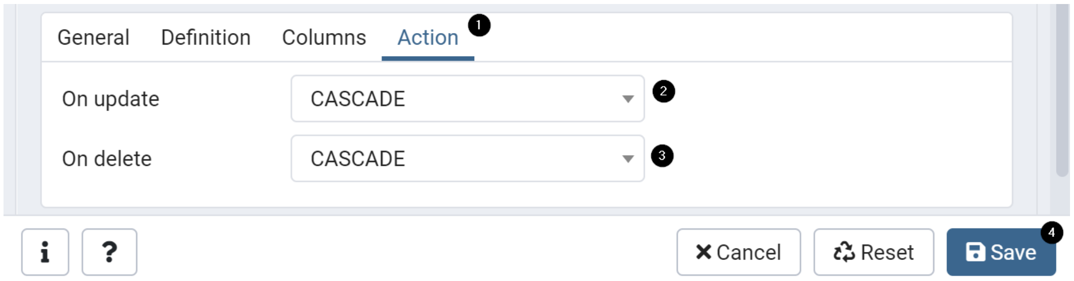
Local	Referenced	Referenced Table
manufacturerid	id	factory.manufacturer

Figure 45: Creating a 1:n relationship for the manufacturer:product

39. Now add an **`INSERT And UPDATE`** rule, **`Cascade on DELETE`**. This way when a manufacturer gets deleted, all the associated products will be deleted as well. To do so, switch to the **`Action`** tab (in the same window as shown in *Figure 46*) (**1**).

40. Then select **`CASCADE`** for both **`On update`** (**2**) and **`On delete`** (**3**).

41. Finally, hit the **Save** button (**4**):

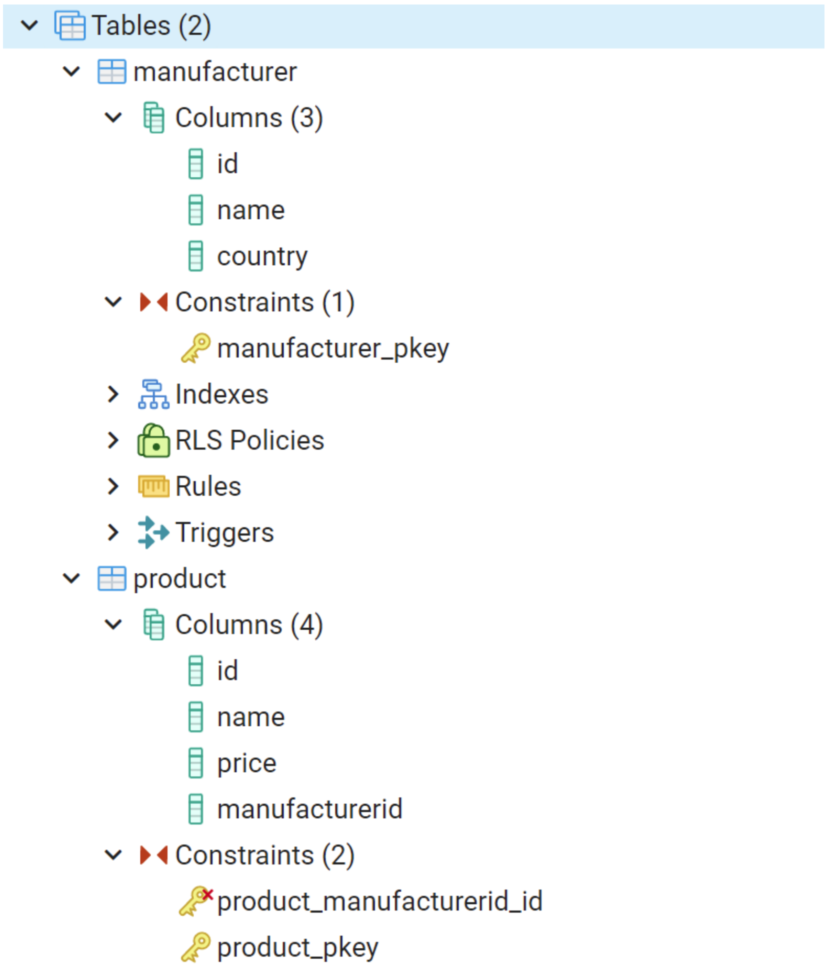


On update CASCADE

On delete CASCADE

Figure 46: Rule set to delete all products when the manufacturer column is removed

42. Once everything is saved, refresh your tables and see the following *Figure 47*:



Tables (2)
manufacturer
Columns (3)
id
name
country
Constraints (1)
manufacturer_pkey
Indexes
RLS Policies
Rules
Triggers
product
Columns (4)
id
name
price
manufacturerid
Constraints (2)
product_manufacturerid_id
product_pkey

Figure 47: Two tables—product and manufacturer—with columns and constraints set

You have just made your first steps in database development. This was a basic example of how databases, tables, and the relationships between them are created, which is merely the tip of the iceberg compared to the complexity and work that big enterprise projects require. However, it's a great start.

> **NOTE**
>
> The steps of this *Exercise 1* can also be done using P-SQL statements as shown on GitHub at https://packt.link/H0eNc. Also, as per .NET 6.0, the **`main()`** method is not compulsory for **`Program.cs`**.

Now that you have practiced creating a new database factory with products and manufacturers, you can move on to the next exercise where you'll be working with data in the new tables you've just added.

EXERCISE 2: CREATING, UPDATING, DELETING, AND READING MANUFACTURERS AND PRODUCTS

Your team was happy with the initial database design. Now, you need to populate the database based on your knowledge of how the data could look. There is a term for common operations of a database named **CRUD** (**Create, Read, Update, Delete**) operations. In this exercise, you will write SQL to implement the following:

- Create two manufacturers and three products.
- Retrieve all the products and manufacturers.
- Update the record for one product.
- Delete the record for one manufacturer.

The following steps will help you to complete this exercise:

1. Select **`globalfactory2021`** (**1**) and open the **`Query tool`** button:

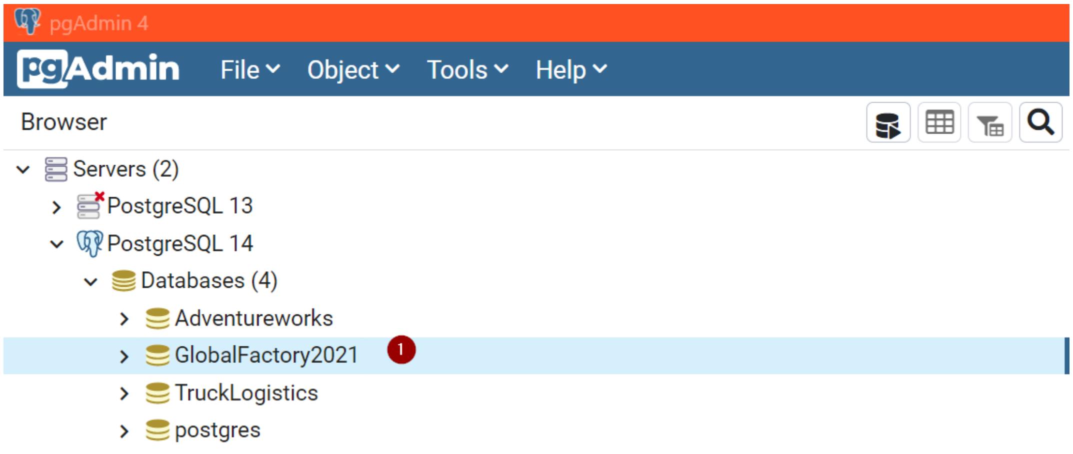


Figure 48: globalfactory2021 database displayed in pgAdmin

2. Adding two manufacturers is just a matter of executing two **`INSERT`** statements. Type the following in the Query Editor window:

```
INSERT INTO factory.manufacturer (name, country) VALUES ('Belgium
Toys', 'Belgium');
INSERT INTO factory.manufacturer (name, country) VALUES
('Wonderland', 'USA');
INSERT INTO factory.product (name, price, manufacturerid) VALUES
('Toy Car', 1.50, 1);
INSERT INTO factory.product (name, price, manufacturerid) VALUES
('Firefighter Truck', 10.49, 1);
INSERT INTO factory.product (name, price, manufacturerid) VALUES
('Toy Rocket', 7.99, 2);
```

Note the usage of semi-colons at the end of each query in the preceding snippet. This is required when you run multiple queries. Upon executing these statements, you should have a **`manufacturer`** named **`Belgium Toys`**, with two **`product`**—**`Toy Car`** and **`Firefighter Truck`**. Additionally, you should see another manufacturer named **`Wonderland`** with just one product—**`Toy Rocket`**.

3. To get everything that you have added, you will need an **`INNER JOIN`** method. You are choosing this option because a **`product`** must have a **`manufacturer`**. It is not optional. Type the following:

```
SELECT product.name, product.price, manufacturer.name as
manufacturer, manufacturer.country
FROM factory.product as product
INNER JOIN factory.manufacturer as manufacturer
ON product.manufacturerid = manufacturer.id
```

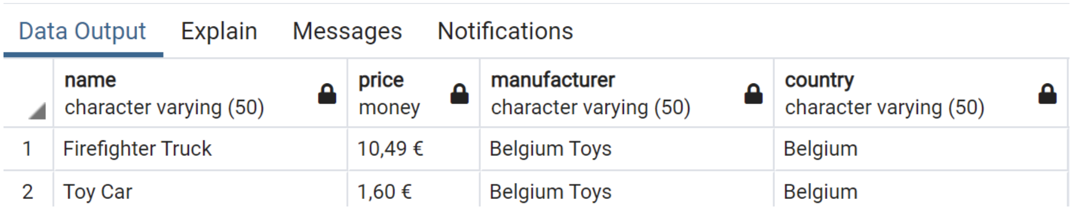
4. Write an **`UPDATE`** SQL statement to modify the **`Toy Car`** price:

```
UPDATE factory.product SET price = 1.6
WHERE name = 'Toy Car'
```

5. Run a **`DELETE`** statement to remove the **`Wonderland`** manufacturer:

```
DELETE FROM factory.manufacturer
WHERE name = 'Wonderland'
```

6. Now, select everything from both tables (run the query from *Step 4*), and you should see the following output as shown in *Figure 49*:



	name character varying (50)	price money	manufacturer character varying (50)	country character varying (50)
1	Firefighter Truck	10,49 €	Belgium Toys	Belgium
2	Toy Car	1,60 €	Belgium Toys	Belgium

Figure 49: Applying CRUD operations on product and manufacturer database

> **NOTE**
>
> The steps of *Exercise 2* can also be done using P-SQL statements as shown on the GitHub at https://packt.link/0qjBh.

This and the previous exercise give us a complete view of what designing and working with a simple database could look like.

SUMMARY

Persisting and querying data has a lot to do with optimizing your database structure. This reference chapter on simple Databases and SQL has taught you how to create tables and write SQL queries for CRUD operations. In the next *Chapter 6*, *Entity Framework with SQL Server*, you will learn to rapidly prototype databases and integrate them using C#.