## Query to Display User Tables

**A user-defined table** is a representation of defined information in a table, and they can be used as arguments for procedures or **user-defined functions**. Because they’re so useful, it’s useful to keep track of them using the following query.

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Sys.objects WHERE Type='u' |

## 15. Query to Display Primary Keys

A primary key uniquely identifies all values within a table. The following SQL query lists all the fields in a **table’s primary key.**

|  |  |
| --- | --- |
| 1 | SELECT \* from Sys.Objects WHERE Type='PK' |

## 16. Query for Displaying Unique Keys

**A Unique Key** allows a column to ensure that all of its values are different.

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Sys.Objects WHERE Type='uq' |

## 17. Displaying Foreign Keys

**Foreign keys link** one table to another – they are attributes in one table which refer to the primary key of another table.

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Sys.Objects WHERE Type='f' |

## 18. Displaying Triggers

A **Trigger** is sort of an ‘event listener’ – i.e, it’s a pre-specified set of instructions that execute when a certain event occurs. The **list of defined triggers** can be viewed using the following query.

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Sys.Objects WHERE Type='tr' |

## 19. Displaying Internal Tables

**Internal tables** are formed as a by-product of a **user-action** and are usually not accessible. The data in internal tables cannot be manipulated; however, the metadata of the internal tables can be viewed using the following query.

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Sys.Objects WHERE Type='it' |

## 20. Displaying a List of Procedures

A stored procedure is a **group of SQL queries** that logically form a single unit and perform a particular task. Thus, using the following query you can keep track of them:

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Sys.Objects WHERE Type='p' |

## Inner Join

Let's say that you have a music database. Albums, artist, genres, reviews, etc. are all stored in separate tables. If you want to get all records from two tables that are related, you can use a simple **inner join**. This selects data where the values match in both tables. Check out these tables. The first one is some sample data and the second one is the albums table:

#### Sample Data

|  |  |
| --- | --- |
| **artistID** | **artistName** |
| 1 | Journey |
| 2 | Meat Loaf |
| 3 | Enya |
| 4 | Kate Wolf |
| 5 | Aerosmith |

#### Albums table

|  |  |  |
| --- | --- | --- |
| **albumID** | **artistID** | **albumTitle** |
| 1 | 1 | Raised on Radio |
| 2 | 1 | Greatest Hits |
| 3 | 2 | Bat out of Hell |
| 4 | 2 | Dead Ringer |
| 5 | 3 | The Celts |
| 6 | 4 | Poet's Heart |

This code is how the inner join would appear:

1. SELECT \*
2. FROM tblAlbum
3. INNER JOIN tblArtist ON
4. tblAlbum.artistID = tblArtist.artistID;

Now, with this new table we can see that the result is 6 rows because we only care about those records where the IDs match.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **albumIID** | **tblAlbum.artistID** | **albumTitle** | **tblArtist.artistID** | **artistName** |
| 1 | 1 | Raised on Radio | 1 | Journey |
| 2 | 2 | Greatest Hits | 1 | Journey |
| 3 | 2 | Bat out of Hell | 2 | Meat Loaf |
| 4 | 2 | Dead Ringer | 2 | Meat Loaf |
| 5 | 3 | The Celts | 3 | Enya |
| 6 | 4 | Poet's Heart | 4 | Kate Wolf |

Now, here is a graphic representing an inner join. Looks pretty simple, right?

|  |
| --- |
| SQL inner join graphic |

## Outer Join

Now, let's take a closer look at using the outer join method. A **left outer join** returns ALL matching records in the left-hand table, plus the records that match in the middle (the inner join).

The SQL statement is as follows:

1. SELECT \* FROM tblArtist
2. LEFT OUTER JOIN tblAlbum ON
3. tblArtist.artistID = tblAlbum.artistID;

This generates the seven rows because now we get the record for Aerosmith. This band doesn't have any albums entered yet, since there are so many to add. However, when running the left outer join, we get to see their entry.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **tblArtist.artistID** | **artistName** | **albumID** | **tblAlbum.artistID** | **albumTitle** |
| 1 | Journey | 1 | 1 | Raised on Radio |
| 1 | Journey | 2 | 1 | Greatest Hits |
| 2 | Meat Loaf | 3 | 2 | Bat out of Hell |
| 2 | Meat Loaf | 4 | 2 | Dead Ringer |
| 3 | Enya | 5 | 3 | The Celts |
| 4 | Kate Wolf | 6 | 4 | Poet's Heart |
| 5 | Aerosmith |  |  |  |

And now, here is a graphic that represents the left outer join:

|  |
| --- |
| SQL left outer join graphic |

### Right Outer Join

Now, let's take a look at the right outer join method. Let's say that you had an album called Unknown, and it wasn't yet tied to an artist quite yet. To retrieve this data element, you can use the **right outer join**. This is like the left outer, except you retrieve matching records plus those in the right-hand table.

Here is the SQL statement:

1. SELECT \*
2. FROM tblArtist RIGHT OUTER JOIN tblAlbum ON tblArtist.artistID = tblAlbum.artistID;

Here is a graphic that shows a right outer join:

|  |
| --- |
| SQL right outer join graphic |

### Full Outer Join (Rows That Don't Join)

Now let's take a look a full outer join, in which we have rows that don't join. Consider that we've inserted some new albums and some new artists in the database, but we haven't joined them together. You can see this playing out in the code:

1. SELECT \* FROM tblArtist
2. FULL OUTER JOIN tblAlbum ON
3. tblArtist.artistID = tblAlbum.albumID
4. WHERE
5. tblArtist.artistID = NULL or tblAlbum.artistID = NULL;