**The Relational Database Model**

**A Logical View of Data**

The *relational model* is a way to structure and manage data in a database, which was introduced by E.F. Codd in 1970. Let me break down the key points you provided:

1. **View data logically rather than physically**:
   * In the relational model, the data is represented logically, not by how it is physically stored on the disk. This means users and developers interact with the data based on its structure and relationships (using tables and fields), without needing to know the physical details of how the data is stored (e.g., memory locations or data retrieval mechanisms). This abstraction simplifies working with data.
2. **Table**:
   * The core element of the relational model is the *table* (also called a *relation*). Each table consists of rows and columns. Rows represent individual records, and columns represent attributes of those records. For example, a table might represent a list of customers, where each row corresponds to a single customer and each column contains information like their name, address, or phone number.
3. **Relational database model is easier to understand**:
   * Compared to other data models (like hierarchical or network models), the relational model is generally easier to work with because data is structured in a simple, intuitive table format. Relationships between data are represented through the use of keys (like primary keys and foreign keys), which allows users to understand and query the data without having to manage complex structures.

**Tables and Their Characteristics**

1. **Logical view of relational database is based on relation**:
   * In a relational database, the *logical view* is how users or developers see and interact with the data. This view is based on the concept of a *relation*, which is the foundational structure in a relational database. A *relation* is essentially what we call a *table*, and it consists of data organized into rows and columns. The term "logical view" refers to the way data is represented conceptually (through tables) rather than how it's physically stored on disk. This view hides the complexities of how the database manages, retrieves, and stores the data.
2. **Table: two-dimensional structure composed of rows and columns**:
   * A *table* is a two-dimensional structure in a relational database. The table is composed of:
     + **Rows**: Each row (also called a *record* or *tuple*) represents a single, individual entry in the table. For instance, in a "Customers" table, each row could represent a different customer.
     + **Columns**: Each column (also called an *attribute* or *field*) represents a specific characteristic or piece of data for every record in the table. For example, in a "Customers" table, columns might include "Customer ID," "Name," "Email," etc.

**Example:**

Imagine a table named Customers:

| **CustomerID** | **Name** | **Email** | **Phone** |
| --- | --- | --- | --- |
| 1 | Alice | alice@email.com | 555-1234 |
| 2 | Bob | bob@email.com | 555-5678 |

* **Rows**: In this case, each row represents a specific customer (Alice, Bob).
* **Columns**: The columns specify the type of information stored for each customer (ID, Name, Email, Phone).

This two-dimensional table structure makes it easy to visualize and query data in a relational database, where each row is a distinct record and each column represents an attribute of those records. This simplicity and organization are part of what makes the relational model effective and user-friendly.

**Keys**

This statement highlights two essential concepts in the relational database model: **unique identification of rows** and the role of **keys**. Let me break it down:

**1. Each row in a table must be uniquely identifiable:**

* In a relational database table, each row (or record) represents a unique instance of the data, such as a specific customer, employee, or order. To distinguish one row from another, there must be a unique identifier for each row.
* This is important because without a unique way to identify each row, there could be confusion when trying to retrieve, update, or delete specific records.
* This unique identifier is typically provided by a *key*—most commonly a **primary key**.

**Example**:  
In a table named Employees, each employee must have a unique identifier, like an **EmployeeID**. Two employees could have the same name, but their IDs would be unique.

| **EmployeeID** | **Name** | **Department** |
| --- | --- | --- |
| 1001 | John | Sales |
| 1002 | Jane | Marketing |
| 1003 | John | IT |

Here, EmployeeID ensures that each row is uniquely identifiable, even if multiple employees share the same name.

**2. Key: one or more attributes that determine other attributes:**

* A **key** is a column (or a set of columns) in a table that is used to uniquely identify rows and determine other attributes in that row.
* There are different types of keys in relational databases, but the two most important types are:
  + **Primary Key**: A single column or combination of columns that uniquely identifies each row in the table. For instance, in the Employees table, EmployeeID can serve as the primary key.
  + **Foreign Key**: A column that links one table to another, establishing a relationship between the two tables.

The idea of a key "determining other attributes" means that knowing the value of the key should allow you to determine or retrieve the values of other columns in the row.

**Example**:  
In the Employees table, EmployeeID can be considered a primary key. If you know the EmployeeID, you can determine the employee's other attributes like Name and Department.

| **EmployeeID** | **Name** | **Department** |
| --- | --- | --- |
| **1001** | John | Sales |

If you know EmployeeID = 1001, you can look up the values for Name = John and Department = Sales.

**Types of Keys:**

**1. Composite Key:**

* A **composite key** is a key made up of two or more attributes (columns) that together uniquely identify a row in a table.
* This is necessary when no single attribute is sufficient to uniquely identify a record, but a combination of attributes can achieve that.

**Example**:  
Consider a table named Orders, where a customer can place multiple orders on the same day, and the same product may be ordered by different customers. To uniquely identify each order, you need to use both CustomerID and OrderDate together as a composite key.

| **CustomerID** | **OrderDate** | **Product** | **Quantity** |
| --- | --- | --- | --- |
| 101 | 2024-09-15 | Laptop | 1 |
| 101 | 2024-09-15 | Mouse | 2 |
| 102 | 2024-09-15 | Keyboard | 1 |

Here, CustomerID alone is not enough to uniquely identify the order, and neither is OrderDate, but together (CustomerID + OrderDate), they form a composite key that can uniquely identify each row.

**2. Key Attribute:**

* A **key attribute** is any attribute (column) that is part of a key (whether it's a single attribute key or a composite key). If the attribute is used to help uniquely identify records, it's considered a key attribute.

**Example**:  
In the example above, both CustomerID and OrderDate are key attributes because together they form the composite key for the Orders table.

**3. Superkey:**

* A **superkey** is any set of attributes (one or more) that can uniquely identify each row in a table. This can include the primary key, candidate keys, or even a key that contains extra, unnecessary attributes.

**Example**:  
In a table Employees, EmployeeID alone is enough to uniquely identify each employee, so it's a superkey. However, a combination of EmployeeID and another column like Name would still be a superkey, but with unnecessary attributes (since EmployeeID alone can do the job).

| **EmployeeID** | **Name** | **Department** |
| --- | --- | --- |
| 1001 | John | Sales |
| 1002 | Jane | Marketing |

* EmployeeID is a superkey.
* EmployeeID + Name is also a superkey, but it contains an unnecessary attribute (Name), because EmployeeID alone uniquely identifies the row.

**4. Candidate Key:**

* A **candidate key** is a minimal superkey. It’s a superkey with no unnecessary attributes. In other words, it's a key that can uniquely identify each row with the fewest possible attributes.
* A table can have more than one candidate key, and one of these candidate keys is usually chosen as the **primary key**.

**Example**:  
In the Employees table:

| **EmployeeID** | **Name** | **SSN** |
| --- | --- | --- |
| 1001 | John | 123-45-6789 |
| 1002 | Jane | 987-65-4321 |

* EmployeeID is a candidate key because it uniquely identifies each row without extra attributes.
* SSN (Social Security Number) could also be a candidate key because it also uniquely identifies each employee.

Since both EmployeeID and SSN can uniquely identify employees, they are candidate keys. However, only one would be chosen as the primary key (e.g., EmployeeID).

**1. Entity Integrity:**

* **Entity integrity** ensures that each row (or **entity instance**) in a table has a unique identity, which is typically guaranteed by the **primary key**. This means no two rows in a table should have the same primary key value.
* The primary key must be unique and cannot be NULL (i.e., it must always have a value), ensuring that every row can be distinctly identified.

**Example**:  
Consider a Students table where each student is identified by a unique StudentID:

| **StudentID** | **Name** | **Age** |
| --- | --- | --- |
| **1001** | Alice | 20 |
| **1002** | Bob | 22 |
| **1003** | Charlie | 19 |

* The StudentID column ensures that each student has a unique identifier, preventing duplicate entries. No two rows can have the same StudentID, and the StudentID column cannot be left blank (i.e., no NULL values are allowed). This enforces **entity integrity**.

**2. Nulls:**

* A **null** represents the absence of data in a database. It means no value has been assigned to a particular column for a given row.
* **Nulls are not permitted in the primary key**: Since the primary key is responsible for uniquely identifying each row, allowing nulls in the primary key would violate this uniqueness requirement. If a primary key were null, it would be impossible to uniquely identify the row.
* **Nulls should be avoided in other attributes**: Although nulls can be allowed in non-primary key columns, they should be used sparingly. Relying too much on nulls can lead to complications in data processing, querying, and understanding the data. For example, you may not know if a value is missing due to an error or if it’s simply unknown.

**Example**: Let’s extend the Students table:

| **StudentID** | **Name** | **Age** | **Email** |
| --- | --- | --- | --- |
| **1001** | Alice | 20 | alice@domain.com |
| **1002** | Bob | 22 | **NULL** |
| **1003** | Charlie | 19 | charlie@domain.com |

* **Primary Key**: The StudentID cannot be null (it must always have a value), so each student can be uniquely identified. This is part of enforcing **entity integrity**.
* **Nulls in other attributes**: In this case, Bob’s email is NULL, meaning no data was entered for that column. While it's acceptable to leave some non-essential fields null, such as Email, using too many nulls can make it hard to interpret the data and could create challenges in queries (e.g., how do you handle missing values?).

**1. Foreign Key (FK):**

* A **foreign key** is an attribute (or set of attributes) in a table that links to the **primary key** of another table. The foreign key establishes a relationship between the two tables, enforcing a connection where data in one table relates to data in another.
* The foreign key's values must match the primary key values in the related table.

**Example**:  
Consider two tables: Orders and Customers. The Customers table contains the list of customers, and Orders records the orders they place. To link an order to the correct customer, the CustomerID in the Orders table would be the **foreign key**, which refers to the CustomerID in the Customers table (the primary key there).

**Customers Table**:

| **CustomerID** | **Name** | **Email** |
| --- | --- | --- |
| **101** | Alice | alice@email.com |
| **102** | Bob | bob@email.com |
| **103** | Charlie | charlie@email.com |

**Orders Table**:

| **OrderID** | **CustomerID (FK)** | **Product** | **Quantity** |
| --- | --- | --- | --- |
| 1001 | **101** | Laptop | 1 |
| 1002 | **103** | Mouse | 2 |
| 1003 | **102** | Keyboard | 1 |

In this example, CustomerID in the Orders table is the **foreign key** that links to CustomerID in the Customers table, ensuring each order is associated with a valid customer.

**2. Referential Integrity:**

* **Referential integrity** ensures that the foreign key in one table always refers to an existing, valid row in the related table. This rule prevents "orphan" records (rows with foreign key values that don't have corresponding rows in the related table).
* It maintains consistency by ensuring that if a foreign key contains a value, that value must exist as a primary key in the related table.

**Example**:  
Continuing with the Customers and Orders tables:

If CustomerID in the Orders table contains a value, that value must exist in the Customers table. This enforces **referential integrity**. For example:

* In the Orders table, CustomerID = 101 is valid because there is a CustomerID = 101 in the Customers table (referring to Alice).
* If an order had CustomerID = 999 but there was no corresponding CustomerID = 999 in the Customers table, this would violate **referential integrity** because the foreign key would be pointing to a non-existent customer.

Referential integrity rules ensure this doesn't happen by either rejecting invalid foreign key values or enforcing cascading updates/deletes (depending on how the system is set up).

**3. Secondary Key:**

* A **secondary key** is any attribute or set of attributes that is not a primary key but can be used for **data retrieval**. It is used to index or search for records efficiently based on non-primary attributes.
* Secondary keys do not need to be unique and are not used for enforcing table relationships like primary or foreign keys. Instead, they help optimize querying for faster data retrieval.

**Example**:  
In the Customers table:

| **CustomerID** | **Name** | **Email** | **Phone** |
| --- | --- | --- | --- |
| 101 | Alice | alice@email.com | 555-1234 |
| 102 | Bob | bob@email.com | 555-5678 |
| 103 | Charlie | charlie@email.com | 555-9999 |

* Here, CustomerID is the **primary key** (unique identifier), but the Email field can be designated as a **secondary key** for searching or retrieving customers based on their email address.
* For example, if the system needs to retrieve a customer by their email (alice@email.com), the Email column serves as the **secondary key** to speed up the search process.

**Relational algebra** is a theoretical framework used in databases for manipulating and querying data. It defines a set of **relational operators** that can be used to operate on **relations** (i.e., tables) and produce new relations (new tables). These operators allow users to perform various operations like filtering, combining, and transforming data in a structured way.

Let’s break down each of the main **relational algebra operators** and explain them with examples:

**1. SELECT (σ):**

* **Purpose**: Used to filter rows from a table (relation) based on a condition. It retrieves tuples (rows) that satisfy a specified condition.
* **Example**:  
  Consider a Students table:

| **StudentID** | **Name** | **Age** | **Major** |
| --- | --- | --- | --- |
| 101 | Alice | 20 | Computer Science |
| 102 | Bob | 22 | Mathematics |
| 103 | Charlie | 19 | Physics |

* **Operation**: Select students who are older than 20.
* σ Age > 20 (Students)
* **Result**:

| **StudentID** | **Name** | **Age** | **Major** |
| --- | --- | --- | --- |
| 102 | Bob | 22 | Mathematics |

**2. UNION (∪):**

* **Purpose**: Combines two relations (tables) by including all rows that appear in either table, eliminating duplicates.
* **Example**:  
  Two tables, CS\_Students and Math\_Students:

**CS\_Students**:

| **StudentID** | **Name** |
| --- | --- |
| 101 | Alice |
| 102 | Bob |

**Math\_Students**:

| **StudentID** | **Name** |
| --- | --- |
| 102 | Bob |
| 103 | Charlie |

**Operation**: Find all students in either Computer Science or Mathematics.

CS\_Students ∪ Math\_Students

**Result** (duplicates removed):

| **StudentID** | **Name** |
| --- | --- |
| 101 | Alice |
| 102 | Bob |
| 103 | Charlie |

**3. PROJECT (π):**

* **Purpose**: Used to retrieve specific columns (attributes) from a table, eliminating duplicates in the result.
* **Example**:  
  Given the Students table:

| **StudentID** | **Name** | **Age** | **Major** |
| --- | --- | --- | --- |
| 101 | Alice | 20 | Computer Science |
| 102 | Bob | 22 | Mathematics |
| 103 | Charlie | 19 | Physics |

* **Operation**: Get only the names of students.
* π Name (Students)
* **Result**:

| **Name** |
| --- |
| Alice |
| Bob |
| Charlie |

**4. DIFFERENCE (−):**

* **Purpose**: Finds the rows that are in one relation but not in the other.
* **Example**:  
  Given the CS\_Students and Math\_Students tables from the UNION example:

**Operation**: Find students who are only in Computer Science and not in Mathematics.

CS\_Students − Math\_Students

**Result**:

| **StudentID** | **Name** |
| --- | --- |
| 101 | Alice |

**5. JOIN (⨝):**

* **Purpose**: Combines two relations (tables) based on a related column, often a foreign key.
* **Example**:  
  Suppose we have two tables: Orders and Customers.

**Customers**:

| **CustomerID** | **Name** |
| --- | --- |
| 101 | Alice |
| 102 | Bob |

**Orders**:

| **OrderID** | **CustomerID** | **Product** |
| --- | --- | --- |
| 201 | 101 | Laptop |
| 202 | 102 | Mouse |
| 203 | 101 | Keyboard |

**Operation**: Combine Customers and Orders tables to show the customer names along with their orders.

Customers ⨝ Orders

**Result**:

| **CustomerID** | **Name** | **OrderID** | **Product** |
| --- | --- | --- | --- |
| 101 | Alice | 201 | Laptop |
| 101 | Alice | 203 | Keyboard |
| 102 | Bob | 202 | Mouse |

**6. PRODUCT (×):**

* **Purpose**: Returns the Cartesian product of two tables. It pairs every row of one table with every row of the other, which often leads to very large result sets.
* **Example**:  
  Consider the tables A and B:

**A**:

| **X** | **Y** |
| --- | --- |
| 1 | A |
| 2 | B |

**B**:

| **Z** |
| --- |
| C |
| D |

**Operation**: Compute the Cartesian product of A and B.

A × B

**Result**:

| **X** | **Y** | **Z** |
| --- | --- | --- |
| 1 | A | C |
| 1 | A | D |
| 2 | B | C |
| 2 | B | D |

**7. INTERSECT (∩):**

* **Purpose**: Returns only the rows that are common to both tables.
* **Example**:  
  Using the CS\_Students and Math\_Students tables:

**Operation**: Find students who are enrolled in both Computer Science and Mathematics.

CS\_Students ∩ Math\_Students

**Result**:

| **StudentID** | **Name** |
| --- | --- |
| 102 | Bob |

**8. DIVIDE (÷):**

* **Purpose**: Divides one relation by another. This operation is used to find rows in one relation that are related to **all** rows in another relation.
* **Example**:  
  Suppose we have two tables: Customers and ProductsBought.

**Customers**:

| **CustomerID** | **ProductID** |
| --- | --- |
| 101 | 201 |
| 101 | 202 |
| 102 | 202 |
| 102 | 203 |

**RequiredProducts**:

| **ProductID** |
| --- |
| 201 |
| 202 |

**Operation**: Find customers who bought **all** products in the RequiredProducts table.

Customers ÷ RequiredProducts

**Result**:

| **CustomerID** |
| --- |
| 101 |

Customer 101 bought all the products in the RequiredProducts table (both 201 and 202), while customer 102 did not, so only 101 is in the result.

Let's explain these three types of **joins** in relational databases—**natural join**, **equijoin**, and **theta join**—with clear examples to demonstrate their differences and use cases.

**1. Natural Join:**

* **Definition**: A **natural join** automatically joins two tables based on all columns with the same name and data type in both tables. It selects rows with matching values in those common columns and combines them into a single row in the result.
* **Key point**: It does not require specifying the join condition because it joins on all matching columns.

**Example**:  
Consider two tables, Employees and Departments:

**Employees**:

| **EmpID** | **DeptID** | **Name** |
| --- | --- | --- |
| 1 | 101 | Alice |
| 2 | 102 | Bob |
| 3 | 101 | Charlie |

**Departments**:

| **DeptID** | **DeptName** |
| --- | --- |
| 101 | HR |
| 102 | IT |

**Operation**: Perform a natural join on Employees and Departments.

Employees ⨝ Departments

**Result**:

| **EmpID** | **DeptID** | **Name** | **DeptName** |
| --- | --- | --- | --- |
| 1 | 101 | Alice | HR |
| 3 | 101 | Charlie | HR |
| 2 | 102 | Bob | IT |

* In this example, the **natural join** matches the DeptID column (common in both tables) and combines rows from both tables where DeptID matches.

**2. Equijoin:**

* **Definition**: An **equijoin** links two tables based on a specific equality condition between columns. It explicitly compares one or more specified columns from each table using an equality operator (=).
* **Key point**: You have to specify the columns that will be compared.

**Example**:  
Using the same Employees and Departments tables:

**Operation**: Perform an equijoin on Employees and Departments using the DeptID column.

SELECT \* FROM Employees E JOIN Departments D ON E.DeptID = D.DeptID

**Result**:

| **EmpID** | **DeptID (E)** | **Name** | **DeptID (D)** | **DeptName** |
| --- | --- | --- | --- | --- |
| 1 | 101 | Alice | 101 | HR |
| 3 | 101 | Charlie | 101 | HR |
| 2 | 102 | Bob | 102 | IT |

* Here, the **equijoin** explicitly compares the DeptID column in both tables (E.DeptID = D.DeptID). Notice that both DeptID columns appear in the result, unlike the **natural join** which removes duplicates from the result.

**3. Theta Join:**

* **Definition**: A **theta join** is similar to an equijoin, but instead of using the equality operator (=), it allows for **other comparison operators** such as <, >, <=, >=, or !=.
* **Key point**: It uses a non-equality condition for joining tables.

**Example**:  
Let’s extend the Employees table with a new Salary column:

**Employees**:

| **EmpID** | **DeptID** | **Name** | **Salary** |
| --- | --- | --- | --- |
| 1 | 101 | Alice | 60000 |
| 2 | 102 | Bob | 55000 |
| 3 | 101 | Charlie | 50000 |

**Departments**:

| **DeptID** | **DeptName** | **MinSalary** |
| --- | --- | --- |
| 101 | HR | 55000 |
| 102 | IT | 50000 |

**Operation**: Perform a theta join where Salary is greater than MinSalary.

SELECT \* FROM Employees E JOIN Departments D ON E.DeptID = D.DeptID AND E.Salary > D.MinSalary

**Result**:

| **EmpID** | **DeptID** | **Name** | **Salary** | **DeptName** | **MinSalary** |
| --- | --- | --- | --- | --- | --- |
| 1 | 101 | Alice | 60000 | HR | 55000 |

* In this **theta join**, we joined the tables using two conditions: E.DeptID = D.DeptID and E.Salary > D.MinSalary. The result only includes the row where Alice’s salary is greater than the department’s minimum salary.

**1. Inner Join:**

* **Definition**: An **inner join** returns only the rows that have matching values in both tables. If there is no match, the row is excluded from the result set.
* **Key point**: Only **matched records** are returned. Rows that don’t have a corresponding match in both tables are excluded.

**Example**:  
Consider two tables, Employees and Departments:

**Employees**:

| **EmpID** | **DeptID** | **Name** |
| --- | --- | --- |
| 1 | 101 | Alice |
| 2 | 102 | Bob |
| 3 | 103 | Charlie |

**Departments**:

| **DeptID** | **DeptName** |
| --- | --- |
| 101 | HR |
| 102 | IT |
| 104 | Sales |

**Operation**: Perform an inner join between Employees and Departments on the DeptID column.

SELECT \* FROM Employees E INNER JOIN Departments D ON E.DeptID = D.DeptID

**Result**:

| **EmpID** | **DeptID** | **Name** | **DeptName** |
| --- | --- | --- | --- |
| 1 | 101 | Alice | HR |
| 2 | 102 | Bob | IT |

* In this **inner join**, only the rows where DeptID is present in both tables are returned (i.e., DeptID = 101 and DeptID = 102).
* The row for **Charlie** (DeptID = 103) is excluded because there’s no corresponding department with DeptID = 103 in the Departments table.
* The department **Sales** (DeptID = 104) is excluded because there are no employees in that department.

**2. Outer Join:**

* **Definition**: An **outer join** returns all rows from one or both tables, depending on the type of outer join, and fills in NULL for missing matches from the other table. There are three types of outer joins:
  1. **Left Outer Join**: Returns all rows from the left table and matched rows from the right table. If no match exists, the result will contain NULL in columns from the right table.
  2. **Right Outer Join**: Returns all rows from the right table and matched rows from the left table. If no match exists, the result will contain NULL in columns from the left table.
  3. **Full Outer Join**: Returns all rows from both tables, with NULL values in either table where no match exists.

**a) Left Outer Join:**

* **Definition**: Returns all rows from the **left table** (Employees), and matched rows from the **right table** (Departments). Unmatched rows from the right table are filled with NULL.
* **Example**:

**Operation**: Perform a left outer join between Employees and Departments.

SELECT \* FROM Employees E LEFT JOIN Departments D ON E.DeptID = D.DeptID

**Result**:

| **EmpID** | **DeptID** | **Name** | **DeptName** |
| --- | --- | --- | --- |
| 1 | 101 | Alice | HR |
| 2 | 102 | Bob | IT |
| 3 | 103 | Charlie | NULL |

* In this **left outer join**, all rows from the Employees table are included. For **Charlie** (DeptID = 103), there’s no matching department, so DeptName is NULL.

**b) Right Outer Join:**

* **Definition**: Returns all rows from the **right table** (Departments), and matched rows from the **left table** (Employees). Unmatched rows from the left table are filled with NULL.
* **Example**:

**Operation**: Perform a right outer join between Employees and Departments.

SELECT \* FROM Employees E RIGHT JOIN Departments D ON E.DeptID = D.DeptID

**Result**:

| **EmpID** | **DeptID** | **Name** | **DeptName** |
| --- | --- | --- | --- |
| 1 | 101 | Alice | HR |
| 2 | 102 | Bob | IT |
| NULL | 104 | NULL | Sales |

* In this **right outer join**, all rows from the Departments table are included. Since there are no employees in the **Sales** department (DeptID = 104), the EmpID and Name columns are NULL for that row.

**c) Full Outer Join:**

* **Definition**: Returns all rows from both tables, with NULL values in columns where no match is found in the other table.
* **Example**:

**Operation**: Perform a full outer join between Employees and Departments.

SELECT \* FROM Employees E FULL OUTER JOIN Departments D ON E.DeptID = D.DeptID

**Result**:

| **EmpID** | **DeptID** | **Name** | **DeptName** |
| --- | --- | --- | --- |
| 1 | 101 | Alice | HR |
| 2 | 102 | Bob | IT |
| 3 | 103 | Charlie | NULL |
| NULL | 104 | NULL | Sales |

* In this **full outer join**, all rows from both tables are included. Rows with no matching values are filled with NULL:
  + **Charlie** (from Employees) has no matching department, so DeptName is NULL.
  + **Sales** (from Departments) has no matching employee, so EmpID and Name are NULL.