Programming Paradigms

At this point, you may be wondering to yourself: *what in the world does programming paradigm even mean?* You're in luck. You're not the only one. *Programming paradigms* are just ways to classify programming languages according to their style.

There are a lot of different styles of programming, and a programming language isn't necessarily bound to one specific style. For now, let's talk about two opposite styles in Ruby, imperative and declarative.

Imperative Programming

The original style of high-level languages, imperative programming just feeds step-by-step instructions for the computer to execute.

```ruby
def imperative_odds(array)
  idx = 0
  odds = []
  while (idx < array.length)
    if array[idx].odd?
      odds << array[idx]
    end
    i += 1
  end
  odds
end
```

Declarative Programming

In contrast, declarative programming describes what you want to achieve, without going into too much detail about how you're going to do it.

```ruby
def declarative_odds(array)
  odds = array.select { |el| el.odd? }
end
```

The given examples are functionally the same, but are fundamentally different in style. Keep in mind, although programming languages like Ruby allow for different styles, certain languages restrict themselves to the guidelines of a specific programming paradigm. In the next reading, you'll find SQL is an example of declarative programming.

Additional Resources

- [LMU overview of programming paradigms][reading]
- [First part of Stanford lecture series][video] [reading]: http://cs.lmu.edu/~ray/notes/paradigms/ [video]: https://www.youtube.com/watch?v=Ps8jOj7diA0

SQL

Databases

The Ruby objects you create during the lifetime of your program will die when it closes. To save (or persist) data, you need to somehow write the data to permanent storage, like the hard disk. We saw some of this in the serialization chapter.

Applications usually also require rich relationships between pieces of data. Consider a blogging system: users have many posts, posts have many tags, users may be following other users, and so on.

Relational databases (also sometimes referred to as RDBMS, relational database management systems) were developed to provide a means of organizing data and their relationships, persisting that data, and querying that data.

Tables

Relational databases organize data in tables.

id	name	age
1	John	22
2	James	24
3	Sally	54
4	Bob	48
5	Lucy	33
6	Mary	98

Each row is a single entity in the table. Each column houses an additional piece of data for that entity.

Every row in a database table will have a primary key which will be its unique identifier in that table row. By convention, the primary key is simply `id`. Most relational database systems have an auto-increment feature to ensure that the primary keys are always unique.

Breaking your domain down into database tables & columns is an important part of developing any application. Each table will house one type of resource: `people`, `houses`, `blog_posts`, etc. The columns in the table will house the data associated with each instance of the resource.

Database Schemas

Your database schema is a description of the organization of your database into tables and columns.

Designing your schema is one of the first and most important steps when writing an application. It forces you to ask a basic but essential question: what data does my application need to function?

When implementing a database schema, you must decide on three things:

- the tables you will have
- the columns each of those tables will have
- and the data type of each of those columns

Schemas are mutable, so the decisions up front are not at all set in stone. Still, you should spend time thinking about your schema at the outset to avoid making major, avoidable mistakes.

The concept of static typing may be new to you. Ruby is dynamically typed -- there is no need to specify in method parameters or variables the class (also called type) of the data stored in it. Ruby won't stop you even if you store something silly like a `Cat` object in a variable named `favorite_dog`, or a `String` in a variable named `number`.

SQL is not quite so flexible; you must specify the type of data that will go into each column.

Here are a few of the most common datatypes:

- `BOOLEAN`
- `INT`
- `FLOAT` (stores "floating point" numbers)
- `VARCHAR(255)` (essentialy a string with a length limit of 255 chars)
- `TEXT` (a string of unlimited length)
- `DATE`
- `DATETIME`
- `TIME`
- `BLOB` (non-textual, binary data; e.g., an image)

We'll see how exactly we create tables, include columns, and specify column types in just a bit.

Modeling Relationships

Now we have a way to store users and additional bits of data on them, but how would we store associated entities like a blog post written by a user?

We probably have the sense that they should be in their own tables since they're not really additional attributes on a user (which would call for additional columns), nor are they users themselves (which would call for additional rows). But if posts were in their own table, how would we know that they were associated with a particular user?

We model these relationships between entries in separate tables through *foreign keys*. A foreign key is a value in a database table whose responsibility is to point to a row in a different table. Check out the posts table below (and pretend that people were a bit more creative in their titles and bodies).

```
posts table:

id |   title   |     body    |  user_id
---------------------------------------
1  |  'XXXX'   |   'xyz...'  |    3
2  |  'XXXX'   |   'xyz...'  |    5
3  |  'XXXX'   |   'xyz...'  |    7
4  |  'XXXX'   |   'xyz...'  |    10
5  |  'XXXX'   |   'xyz...'  |    2
```

```
  6  |  'XXXX'   |   'xyz...'  |    5
```

The `user_id` column is a foreign key column. If we wanted to find all the posts for the user with `id` 5, we'd look in the posts table and retrieve all the posts where the `user_id` column had a value of 5. If you already know a little SQL:

```
SELECT
  *
FROM
  posts
WHERE
  posts.user_id = 5
```

By convention, the foreign key in one table will reference the primary key in another table. We usually call the column that houses the foreign key `[other_table_name_singularized]_id` .

Foreign keys are how we model relationships between pieces of data across multiple tables. This also allows us to ensure that data is not duplicated across our database. Posts live in a single place, users in another, and the foreign key (`user_id`) in `posts` expresses the relation between the one and the other.

Structured Query Language (SQL)

Now that we know what these tables look like and generally how relationships are modeled between them, how do we actually get at the data?

Enter SQL. SQL is a domain-specific language that's designed to query data out of relational databases.

Here's a sample SQL query (we'll break it down in just a second):

```
-- Find crazy cat people
SELECT
  name, age, has_cats
FROM
  tenants
WHERE
  (has_cats = true AND age > 50)
```

SQL queries are broken down into clauses. Here, there is the `SELECT` clause, the `FROM` clause, and the `WHERE` clause. `SELECT` takes a list of comma-separated column names; only these columns of data will be retrieved. `FROM` takes a table name to query. `WHERE` takes a list of conditions separated by `AND` or `OR` ; only rows matching these conditions are returned..

SQL provides powerful filtering with `WHERE` ; it supports the standard comparison and equality operators (`<` , `>` , `>=` , `<=` , `=` , `!=`) as well as boolean operators (`AND` , `OR` , `NOT`).

There are 4 main data manipulation operations that SQL provides:

- `SELECT` : retrieve values from one or more rows
- `INSERT` : insert a row into a table
- `UPDATE` : update values in one or more existing rows
- `DELETE` : delete one or more rows

Below are brief descriptions of each of the operators syntactical signatures and a couple simple examples of their use:

`SELECT`

Structure:

```
SELECT
  one or more columns (or all columns with *)
FROM
  one (or more tables, joined with JOIN)
WHERE
  one (or more conditions, joined with AND/OR);
```

Examples:

```
SELECT
  *
FROM
  users
WHERE
  name = 'Ned';

SELECT
  account_number, account_type
FROM
  accounts
WHERE
  (customer_id = 5 AND account_type = 'checking');
```

INSERT

Structure:

```
INSERT INTO
  table name (column names)
VALUES
  (values);
```

Examples:

```
INSERT INTO
  users (name, age, height_in_inches)
VALUES
  ('Santa Claus', 876, 34);

INSERT INTO
  accounts (account_number, customer_id, account_type)
VALUES
  (12345, 76, 'savings');
```

UPDATE

Structure:

```
UPDATE
  table_name
SET
  col_name1 = value1,
  col_name2 = value2
WHERE
  conditions
```

Examples:

```
UPDATE
  users
SET
  name = 'Eddard Stark', house = 'Winterfell'
WHERE
  name = 'Ned Stark';

UPDATE
  accounts
SET
  balance = 30
WHERE
  id = 6;
```

`DELETE`

Structure:

```
DELETE FROM
  table_name
WHERE
  conditions
```

Examples:

```
DELETE FROM
  users
WHERE
  (name = 'Eddard Stark' AND house = 'Winterfell');

DELETE FROM
  accounts
WHERE
  customer_id = 666;
```

Schema Definitions

Before basic querying can take place though, you need to actually define your database schema. There are three operators that SQL provides to manipulate a database schema:

- `CREATE TABLE`
- `ALTER TABLE`
- `DROP TABLE`

Here's an example of creating a users table (we'll break it down shortly):

```
CREATE TABLE users (
  id INTEGER PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  birth_date DATE,
  house VARCHAR(255),
  favorite_food VARCHAR(20)
);
```

`CREATE TABLE` first specifies the name of the table, and then in parentheses, the list of column names along with their data types.

Querying across multiple tables (JOIN)

Similarly to the objects in a good Ruby program, a well-designed database will split data into tables that each encapsulate some object. Sometimes, we will want to access the data from more than one of these tables at once, but so far we've only seen ways to query a single table. How might we query across tables?

SQL provides a powerful facility: the `JOIN`. A `JOIN` will do just what you'd expect it to do: join together two tables, resulting in a temporary combined table that you can query just like any other. `JOIN` clauses include an `ON` statement, in which you specify how exactly those two tables relate to one another. This is where foreign keys come into play. Check out the simple join below.

Let's write a query that returns the title of all the blog posts written by each user:

```
SELECT
  users.name, posts.title
FROM
  posts
JOIN
  users ON posts.user_id = users.id
```

This will return one row per post, with the user's name appearing next to the title of the post they authored. By storing a `user_id` column in the `posts` table, we can associate user data to posts without adding columns that would duplicate data from other tables in the database and other rows in the `posts` table; we just `JOIN` the tables as needed.

In this example, we joined two different tables using a foreign key stored in a single column. This is the most common case we will see, but `JOIN` is a flexible operator that can handle a variety of conditions. Two variations we will use are self joins, in which we join a table to itself (for example, if we have a table of employees, each of whom has a supervisor in the same table) and joins that use multiple columns to specify the `ON` condition (for example, if we have a bus timetable that identifies routes by a company name and a route number).

In another case, we might have a many-to-many relationship; perhaps, in the example above, `users` can "like" `posts`. In this case, including a foreign key in one of the tables doesn't make sense; a user can like any number of posts, and a post can be liked by any number of users. In this case, we could use a join table that contains a foreign key for each table, allowing us to represent each like with a row linking a user to a post. We would then need two joins to associate users and liked posts, like so:

```
SELECT
  users.name, posts.title
FROM
  posts
JOIN
  likes ON posts.id = likes.post_id
JOIN
  users ON likes.user_id = users.id
```

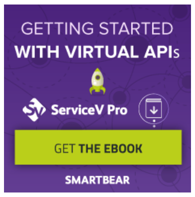
This query will give a list of user names and the posts they have liked.

Resources

- Intro Video

CODING HORROR
programming and human factors

Custom Search



Learn how to transform your software delivery strategy with API virtualization, in this eBook.

Catch Code Errors Before Users Do. Track 5,000 Errors (per month) Free!

ads via Carbon

RESOURCES

About Me
discourse.org
stackexchange.com
Learn Markdown
Recommended Reading

Subscribe in a reader
Subscribe via email

Coding Horror has been continuously published since 2004

11 Oct 2007

A Visual Explanation of SQL Joins

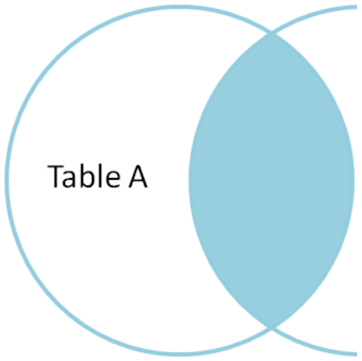
I thought Ligaya Turmelle's post on SQL joins was a great primer for novice developers. Since SQL joins *appear* to be set-based, the use of Venn diagrams to explain them seems, at first blush, to be a natural fit. However, like the commenters to her post, I found that the Venn diagrams didn't quite match the SQL join syntax reality in my testing.

I love the concept, though, so let's see if we can make it work. Assume we have the following two tables. **Table A** is on the left, and **Table B** is on the right. We'll populate them with four records each.

```
id name       id  name
-- ----       --  ----
1  Pirate     1   Rutabaga
2  Monkey     2   Pirate
3  Ninja      3   Darth Vader
4  Spaghetti  4   Ninja
```

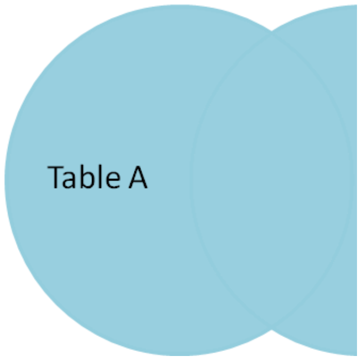
Let's join these tables by the name field in a few different ways and see if we can get a conceptual match to those nifty Venn diagrams.

```
SELECT * FROM TableA
INNER JOIN TableB
ON TableA.name = TableB.name

id  name       id   name
--  ----       --   ----
1   Pirate     2    Pirate
3   Ninja      4    Ninja
```



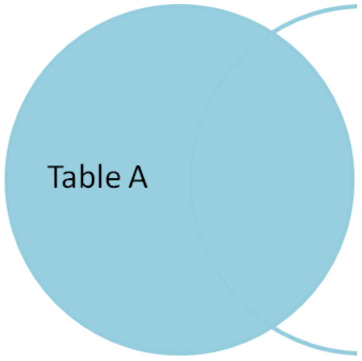
Inner join produces only the set of records that match in both Table A and Table B.

```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name

id    name       id    name
--    ----       --    ----
1     Pirate     2     Pirate
2     Monkey     null  null
3     Ninja      4     Ninja
4     Spaghetti  null  null
null  null       1     Rutabaga
null  null       3     Darth Vader
```



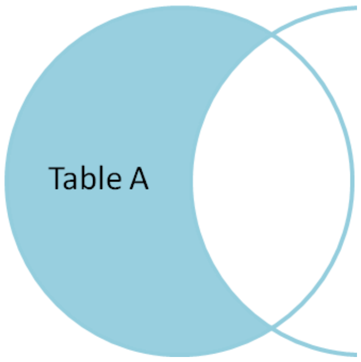
Full outer join produces the set of all records in Table A and Table B, with matching records from both sides where available. If there is no match, the missing side will contain null.

```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name

id  name       id    name
--  ----       --    ----
1   Pirate     2     Pirate
2   Monkey     null  null
3   Ninja      4     Ninja
4   Spaghetti  null  null
```



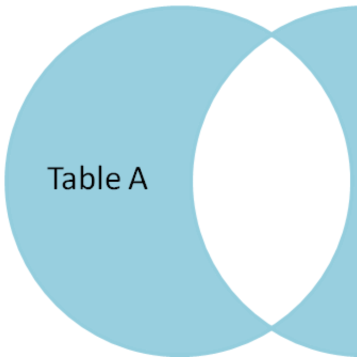
Left outer join produces a complete set of records from Table A, with the matching records (where available) in Table B. If there is no match, the right side will contain null.

```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableB.id IS null

id  name       id    name
--  ----       --    ----
2   Monkey     null  null
4   Spaghetti  null  null
```



To produce the set of records only in Table A, but not in Table B, we perform the same left outer join, then **exclude the records we don't want from the right side via a where clause**.

```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableA.id IS null
OR TableB.id IS null

id    name       id    name
--    ----       --    ----
2     Monkey     null  null
4     Spaghetti  null  null
null  null       1     Rutabaga
null  null       3     Darth Vader
```



To produce the set of records unique to Table A and Table B, we perform the same full outer join, then **exclude the records we don't want from both sides via a where clause**.

There's also a cartesian product or **cross join**, which as far as I can tell, can't be expressed as a Venn diagram:

```
SELECT * FROM TableA
CROSS JOIN TableB
```

This joins "everything to everything", resulting in 4 x 4 = 16 rows, far more than we had in the original sets. If you do the math, you can see why this is a *very* dangerous join to run against large tables.

NEXT
Mouse Ballistics

PREVIOUS
A Lesson in Control Simplicity

This repository Search Pull requests Issues Gist

appacademy / curriculum Private Watch 34 Star 130 Fork 435

Code Issues 2 Pull requests 16 Projects 0 Wiki Pulse Graphs

Branch: master curriculum / sql / readings / self-joins.md Find file Copy path

chungkikelly removed broken db-stack link 7342f0d on Nov 25, 2016

1 contributor

66 lines (54 sloc) 2.75 KB Raw Blame History

Self Joins

What is a self join?

A self join is exactly what it sounds like: an instance of a table joining with itself. The way you should visualize a self join for a given table is by imagining a join performed between two identical copies of that table.

Let's take a look at a classic self-join example. This returns each employee's first and last name along with their manager's.

Employee Table

id	first_name	last_name	manager_id
1	Kush	Patel	NULL
2	Jeff	Fiddler	1
3	Quinn	Leong	2
4	Shamayel	Daoud	2
5	Robert	Koeze	4
6	Munyo	Frey	3
7	Kelly	Chung	4

```
SELECT
  team_member.first_name, team_member.last_name,
   manager.first_name, manager.last_name
FROM
  employee AS team_member
JOIN
  employee AS manager ON manager.id = team_member.manager_id
```

team_member.first_name	team_member.last_name	manager.first_name	manager.last_name
Jeff	Fiddler	Kush	Patel
Quinn	Leong	Jeff	Fiddler
Shamayel	Daoud	Jeff	Fiddler

team_member.first_name	team_member.last_name	manager.first_name	manager.last_name
Robert	Koeze	Shamayel	Daoud
Munyo	Frey	Quinn	Leong
Kelly	Chung	Shamayel	Daoud

In all the examples you've covered thus far, JOINs were performed on two different tables (presumably with two different names), which made it easy to reference a specific column in a table. Since we only deal with one table in a self join, we have to use aliases.

An alias is essentially a nickname for a table (or, in some cases, a column). This is necessary because the query processor needs to make a distinction between the duplicates of the same table to JOIN them. Keep in mind, the keyword AS is not necessary to alias tables or columns. The above SQL query could be rewritten:

```
SELECT
  team_member.first_name, team_member.last_name,
   manager.first_name, manager.last_name
FROM
  employee team_member
JOIN
  employee manager ON manager.id = team_member.manager_id
```

Formatting SQL Code

SQL Conventions

Different programmers use different SQL conventions, but in preparation for ActiveRecord and Rails, which have their own conventions, you should:

- Always name SQL tables snake_case and pluralized. (e.g., `musical_instruments` , `favorite_cats`)
- If a `musician` belongs to a `band` , your `musicians` table will need to store a foreign key that refers to the `id` column in the `bands` table. The foreign key column should be named `band_id` .
- Always have a column named `id` , and use it as the primary key for a table.

You must not write SQL all on a single line. It will be impossible to read:

```
SELECT * FROM table_one LEFT OUTER table_two ON table_one.column_one = table_two.column_x WHERE (table_one.
```

Here's an example of some well formatted SQL code:

```
SELECT
  table_two.column_one,
  table_two.column_two,
  table_two.column_three
FROM
  table_one
LEFT OUTER JOIN
  table_two ON table_one.column_one = table_two.column_x
WHERE
  (table_one.column_three > table_two.column_y
    AND another_condition IS NULL)
GROUP BY
  table_two.column_four
ORDER BY
  table_two.column_four
```

Notice that each component of the SQL statement starts with the keyword aligned left. The body of each component is indented two spaces. Complex `WHERE` clauses are parenthesized and indented two spaces on the following line.

Subqueries

Life gets complicated when you make subqueries. Here's how I do it:

```
SELECT
  bands.*
FROM
  bands
JOIN (
  SELECT
    albums.*
  FROM
    albums
  WHERE
    album.type = "POP"
  GROUP BY
    album.band_id
  HAVING
    COUNT(*) > 3
  ) AS pop_group_albums ON bands.id = pop_group_albums.band_id
WHERE
  band.leader_id IN (
    SELECT
      musicians.id
    FROM
      musicians
    WHERE
      musicians.birth_yr > 1940
  )
```

I put the leading paren on the prior line, indent the query two spaces, and close with a trailing paren at the start of a new line. I put the `ON` of a `JOIN` right after the closing paren.

References

- Based on the style guide How I Write SQL.

(/) **SQLBolt (/)**

Tutorial (/lesson/introduction) More Topics (/topic/introduction)

SQL Topic: Subqueries

You might have noticed that even with a complete query, there are many questions that we can't answer about our data without additional post, or pre, processing. In these cases, you can either make multiple queries and process the data yourself, or you can build a more complex query using SQL subqueries.

Example: General subquery

Lets say your company has a list of all Sales Associates, with data on the revenue that each Associate brings in, and their individual salary. Times are tight, and you now want to find out which of your Associates are costing the company more than the average revenue brought per Associate.

First, you would need to calculate the average revenue all the Associates are generating:

```
SELECT AVG(revenue_generated)
FROM sales_associates;
```

And then using that result, we can then compare the costs of each of the Associates against that value. To use it as a subquery, we can just write it straight into the **WHERE** clause of the query:

```
SELECT *
FROM sales_associates
WHERE salary >
   (SELECT AVG(revenue_generated)
    FROM sales_associates);
```

As the constraint is executed, each Associate's salary will be tested against the value queried from the inner subquery.

A subquery can be referenced anywhere a normal table can be referenced. Inside a **FROM** clause, you can **JOIN** subqueries with other tables, inside a **WHERE** or **HAVING** constraint, you can test expressions against the results of the subquery, and even in expressions in the **SELECT** clause, which allow you to return data directly from the subquery. They are generally executed in the same logical order as the part of the query that they appear in, as described in the last lesson.

Because subqueries can be nested, each subquery must be fully enclosed in parentheses in order to establish proper hierarchy. Subqueries can otherwise reference any tables in the database, and make use of the constructs of a normal query (though some implementations don't subqueries to use **LIMIT** or **OFFSET**).

Correlated subqueries

A more powerful type of subquery is the *correlated subquery* in which the inner query references, and is dependent on, a column or alias from the outer query. Unlike the subqueries above, each of these inner queries need to be run for each of the rows in the outer query, since the inner query is dependent on the current outer query row.

Example: Correlated subquery

Instead of the list of just Sales Associates above, imagine if you have a general list of Employees, their departments (engineering, sales, etc.), revenue, and salary. This time, you are now looking across the company to find the employees who perform worse than average in their department.

For each employee, you would need to calculate their cost relative to the average revenue generated by all people in their department. To take the average for the department, the subquery will need to know what department each employee is in:

```
SELECT *
FROM employees
WHERE salary >
   (SELECT AVG(revenue_generated)
    FROM employees AS dept_employees
    WHERE dept_employees.department = employees.department);
```

These kinds of complex queries can be powerful, but also difficult to read and understand, so you should take care using them. If possible, try and give meaningful aliases to the temporary values and tables. In addition, correlated subqueries can be difficult to optimize, so performance characteristics may vary across different databases.

Existence tests

When we introduced **WHERE** constraints in Lesson 2: Queries with constraints (/lesson/select_queries_with_constraints), the **IN** operator was used to test whether the column value in the current row existed in a fixed list of values. In complex queries, this can be extended using subqueries to test whether a column value exists in a dynamic list of values.

Select query with subquery constraint

```
SELECT *, …
FROM mytable
WHERE column
    IN/NOT IN (SELECT another_column
               FROM another_table);
```

When doing this, notice that the inner subquery must select for a column value or expression to produce a list that the outer column value can be tested against. This type of constraint is powerful when the constraints are based on current data.

Next – SQL Topic: Unions, Intersections & Exceptions (/topic/set_operations)
Previous – Intermediate SQL Topics (/topic/introduction)

NULL and Ternary Logic in SQL

SQL uses ternary logic. This means that a conditional statement can evaluate to `TRUE`, `FALSE` or `NULL` (unknown). Whaaaa? And somehow `NULL` is still 'falsy'? Unfortunately, this won't be the only time you run into logic that defies intuition. *Stay tuned for Javascript quirks.*

If we ask if a `NULL value == NULL`, we will always get false. This is because `NULL` was derived to represent an unknown value. How can we know if two unknowns are the same? We can't. Given that this sort of comparison doesn't yield any useful information, always use `IS NULL` or `IS NOT NULL` in place of the traditional (`==` or `!=`) comparisons.

PostgreSQL CASE

In this tutorial, we will show you how to use the **PostgreSQL CASE** conditional expression to form conditional query.

General PostgreSQL CASE expression

The PostgreSQL CASE expression is the same as `IF/ELSE` statement in other programing languages. PostgreSQL provides two forms of the `CASE` expression. The following illustrates the general form of the `CASE` statement:

```
CASE
       WHEN condition_1  THEN result_1
      WHEN condition_2  THEN result_2
      [WHEN ...]
      [ELSE result_n]
END
```

In this general form, each condition is an expression that returns a boolean value, either `true` or `false`. If the condition evaluates to true, it returns the result which follows the condition, and [illegible] er `CASE` branches do not process at all. If all conditions evaluate to false, the `CASE` expression w [illegible] rn the result in the `ELSE` part. If you omit the `ELSE` clause, the `CASE` expression will return null.

mass if the rental rate is 0.99

economic if the rental rate is 1.99

luxury if the rental rate is 4.99

And you want to know the number of films that belongs mass, economic or luxury price segment. In this case, you can use the `CASE` expression to construct the query as follows:

```
SELECT
 SUM (
 CASE
 WHEN rental_rate = 0.99 THEN
 1
 ELSE
 0
 END
 ) AS "Mass",
 SUM (
 CASE
 WHEN rental_rate = 2.99 THEN
 1
 ELSE
 0
 END
 ) AS "Economic",
 SUM (
 CASE
 WHEN rental_rate = 4.99 THEN
 1
 ELSE
 0
 END
 ) AS "Luxury"
FROM
 film;
```

We used CASE expression to return 1 or 0 if the rental rate meets the requirement. And then we applied SUM function to calculate the total of films for each price segment.

Simple PostgreSQL CASE expression

PostgreSQL provides another form of the `CASE` expression called simple form as follows:

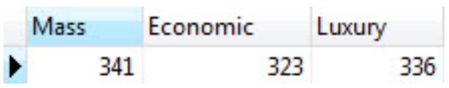
```
CASE expression
WHEN value_1 THEN
 result_1
WHEN value_2 THEN
 result_2
[WHEN ...]
ELSE
 result_n
END;
```

PostgreSQL first evaluates the expression and then compare it with each value in the `WHEN` clause until it finds the value that matches the returned value of the expression. If PostgreSQL does not find any matches, it returns the `result_n` in the `ELSE` part, or `NULL` value if the `ELSE` part is omitted. This is similar to the `switch` statement in other programming languages such as C/C++, Java, etc.

We can rewrite the general `CASE` expression using the simple one as follows:

```
SELECT
 SUM (
 CASE rental_rate
 WHEN 0.99 THEN
 1
 ELSE
 0
 END
 ) AS "Mass",
 SUM (
 CASE rental_rate
 SUM (
 CASE rental_rate
 WHEN 4.99 THEN
 1
 ELSE
 0
 END
 ) AS "Luxury"
FROM
 film;
```

It produces the same result.



Mass	Economic	Luxury
341	323	336

In this tutorial, you have learned how to use both forms of PostgreSQL CASE expression to form complex query.

Related Tutorials

PostgreSQL NULLIF

0 0 0 0

Previous Tutorial:
PostgreSQL SUM Function

Next Tutorial:
PostgreSQL COALESCE

PostgreSQL COALESCE

Summary: in this tutorial, you will learn about the PostgreSQL COALESCE function that returns the first non-null argument. You will learn how to apply this function in SELECT statement to handle null values effectively.

PostgreSQL COALESCE function syntax

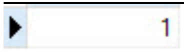
The syntax of the `COALESCE` function is as follows:

```
COALESCE (argument_1, argument_2, …);
```

The `COALESCE` function accepts unlimited number of arguments. It returns the first argument that is not null. If all arguments are null, the `COALESCE` function will return null.

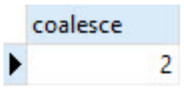
The `COALESCE` function evaluates arguments from left to right until it finds the first non-null argument. All the remaining arguments from the first non-null argument are not evaluated.

The `COALESCE` function provides the same functionality as `NVL` or `IFNULL` function provided by SQL-standard. MySQL has IFNULL function, while Oracle provides `NVL` function.



```
1
```

```
SELECT
 COALESCE (NULL, 2 , 1);
```



```
coalesce
       2
```

We often use the `COLAESCE` function to substitute a default value for null values when we querying the data. For example, we want to display the excerpt of a blog post, if the excerpt is not provided, we can use the first 150 characters of the of the content of the post. To achieve this, we can use the `COALESCE` function as follows:

```
SELECT
 COALESCE (excerpt, LEFT(CONTENT, 150))
FROM
 posts;
```

PostgreSQL COALESCE example

Let's take a look at an example of using `COALESCE` function. First, we create a table named items using CREATE TABLE statement as follows:

```
CREATE TABLE items (
 ID serial PRIMARY KEY,
 product VARCHAR (100) NOT NULL,
 price NUMERIC NOT NULL,
 discount NUMERIC
);
```

There are four fields in the `items` table:

id: the primary key that identify the item in the items table.

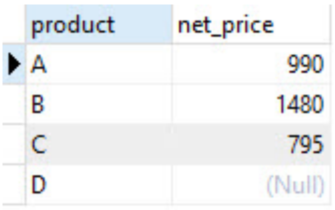
Second, we insert some records into the items table using INSERT statement as follows:

```
INSERT INTO items (product, price, discount)
VALUES
 ('A', 1000 ,10),
 ('B', 1500 ,20),
 ('C', 800 ,5),
 ('D', 500, NULL);
```

Third, we query the net prices of the products using the following formula:

```
net_price = price - discount;
```

```
SELECT
 product,
 (price - discount) AS net_price
FROM
 items;
```



```
product  net_price
A              990
B             1480
C              795
D           (Null)
```

If you look at the fourth row, you will notice that the net price of the product `D` is null which seems not correct. The issue is the discount of the product `D` is null, therefore when we take the null value to calculate the net price, PostgreSQL returns null.

The get the right price, we need to assume that if the discount is null, it is zero. Then we can use the `COALESCE` function as follows:

```
SELECT
 product,
 (price - COALESCE(discount,0)) AS net_price
```

D	500

Now the net price of product `D` is `500` because we use zero instead of null value when we calculated the net price.

Besides using the `COALESCE` function, you can use the CASE expression to handle the null values in this case. See the following query that uses the `CASE` expression to achieve the same result above.

```
SELECT
 product,
 (
 price - CASE
 WHEN discount IS NULL THEN
 0
 ELSE
 discount
 END
 ) AS net_price
FROM
 items;
```

In the query above we say if the discount is null then use zero (0) otherwise use discount value to in the expression that calculate the net price.

In terms of performance, `COALESCE` function and CASE expression are the same. We prefer `COALESCE` function than `CASE` expression because `COALESCE` function makes the query shorter and easier to read.

In this tutorial, you have learned how to use the `COALESCE` function to substitute null values in the query.

Related Tutorials