2022-02-10T18:00:00+02:00
Tihomir Babic

How to Query a Parent-Child Tree in SQL

What are parent-child tree structures in SQL? In this article, we answer that question, talk about query hierarchy, and demonstrate the five most common SQL queries you'll need for these data structures.

Yes, you can use SQL on a parent-child tree structure. I'll show you how in this article. Along the way, I'll walk you through five query examples, starting with the easiest and ending with the most complex. These will use recursive Common Table Expressions (CTEs); if you're not familiar with CTEs or recursion in SQL, I suggest taking our Recursive Queries Course and coming back to this article.

Try our interactive Recursive Queries course. 114 hands-on exercises to help you tackle this advanced concept!

Before we dig into the code, let's look at an overview of the parent-child tree structure and how it is stored in a relational database.

What Is a Parent-Child Tree?

If you understand hierarchical data, you probably know that it's a synonym for a parent-child structure. Both names are very logical; a parent-child tree structure is a set of data structured hierarchically. In other words, there are hierarchical relationships between data items. This means that one data item can be the parent of another data item, which is then called a child. Items are also called tree levels or nodes, and they can assume three main forms:

Root node – The first node, where the parent-child tree starts.

Parent node – This is any node that has one or more descendants (or child) nodes.

(or child) nodes.

Child node – Any node that has a predecessor or parent node.

Real-life examples of parent-child structures include companies' organizational structures (a company consists of departments, the departments consist of teams, and teams consist of employees), family trees (there are parents, children, grandchildren, great-grandchildren, etc.), and natural taxonomies (living things belong to a domain, kingdom, phylum, class, order, family, genus, and species). Even computer folders (C disk, Program Files, Microsoft SQL Server...) , menus (drinks, non-alcoholic beverages, tea...), art and music genres (for example, there was blues, which developed rhythm and blues, which led to soul, funk, etc.), and projects (one project has subprojects, which have tasks, subtasks, etc.) can be considered hierarchical data structures.

Parent-Child Tree Structure in Relational Databases

For SQL to do anything with it, a parent-child tree structure has to be stored in a relational database.

These structures are usually stored in one table with two ID columns, of which one references a parent object ID. That lets us determine the hierarchy between data. In other words, we know which node is a

parent node to which child node and vice versa.

This might sound a little abstract, so I'll show you how it works with a simple example. And I'm going literal with it! My parent-child tree structure will show data about parents and their children. Take a look:

id	first_name	last_name	parent_id
1	Jim	Cliffy	NULL
2	Mark	Cliffy	1
3	Veronica	Cliffy	2

Here, the column `id` shows the child's ID. To find out who that child's parent is, you have to look at the column `parent_id`, find the same ID number in the `id` column, and look in that row for the parent's name.

In other words, Jim Cliffy has no parents in this table; the value in his `parent_id` column is `NULL`. This means he is the root node of this tree structure.

Mark Cliffy is Jim Cliffy's son. How do I know that? Because Mark's `parent_id = 1`, which is Jim Cliffy's ID. Mark Cliffy is a child node, but he's also a parent node. Why? Because Veronica Cliffy is Mark Cliffy's daughter. I know that because her parent has `parent_id = 2`, and the table tells me that's Mark Cliffy. Veronica Cliffy is strictly a child node; she has a parent node, but no child nodes are branching off from her.

Typical Queries Run on a Parent-Child Tree Structure

I'll use the same table for each of these queries. It has the same columns as shown above, only with more rows and different values in them.

Introducing Example Data

The table is named **`parent_child`** and has the following columns:

- `id` – The child's ID and the table's primary key (PK).
- `first_name` – The child's first name.
- `last_name` – The child's last name.

`parent_id` – The child's parent's ID.

Here's the whole table:

id	first_name	last_name	parent_id
1	Rosa	Wellington	NULL
2	Jon	Wellington	1
3	Joni	Wellington	1
4	Marge	Wellington	1
5	Mary	Dijkstra	2
6	Frank	Wellington	2
7	Jason	Wellington	3
8	Bobby	Wellington	4
9	Sammy	Wellington	4
10	Sarah	Wellington	4
11	Sam Francis	Dijkstra	5
12	Stephen	Wellington	6
13	Trent	Wellington	6
14	June	Wellington	9
15	Josephine	Wellington	9
16	Suzy	Wellington	9

You can use this table to check if the queries I'm about to show return the correct output.

Example 1: List All Children of 1 Parent

This is the simplest query, so I'll use it to make you more at ease with the tree structure. Here, I want to find all the children of a specified parent. In this case, I'm interested in finding all children of a person called Marge Wellington, whose ID is 4.

Here's the little query:

```
SELECT first_name,
     last_name
FROM parent_child
WHERE parent_id = 4;
```

I've simply selected the first and the last name from the table and used the `WHERE` clause to show only rows where there is a 4 in the column

`parent_id`.

The result shows three rows:

first_name	last_name
Bobby	Wellington
Sammy	Wellington
Sarah	Wellington

It tells me that Bobby, Sammy, and Sarah Wellington are all Marge Wellington's children. Take a look at the original table, and you'll see that's true.

This was just a warm-up! Let's move to the next one.

Example 2: List a Parent Node For a Child Node

Now, the output in the previous example was a little bit, well, basic. I've listed only the names of the children. It could be really helpful to show the parent's name too. And that's exactly what I'm going to do. I'll show both the child's and parent's first name and last name.

Instead of looking for a parent's children, I'll be now looking for the child's parents. I want to find out who Sam Francis Dijkstra's parent is. Besides the names, I also want to see IDs.

The query for this is:

```
SELECT child.id AS child_id,
     child.first_name AS child_first_name,
     child.last_name AS child_last_name,
     parent.first_name AS parent_first_name,
     parent.last_name AS parent_last_name,
     parent.id AS parent_id

FROM parent_child child
JOIN parent_child parent
  ON child.parent_id = parent.id
WHERE child.id = 11;
```

The main concept I'm introducing here is the self-join. I gave the alias **`child`** to the **`parent_child`** table, and I joined it with itself using the **`parent`** alias. By doing this, I'm acting as I'm working with two different tables. One contains the data about children; that's why I've

different tables. One contains the data about children, that's why I've named it `child`. The other one has data about parents, so I've called it `parent`.

The selected columns reflect that. The children's names and IDs are selected from the "first" table. The parent's names and IDs are selected from the "second" table. The "tables" are joined where `parent_id` equals `id`.

The original table tells me Sam Francis Dijkstra's ID is 11. I've used the `WHERE` clause to filter data and show only Sam Francis's parent. You can also use the `WHERE` clause on the columns `child.first_name` and `child.last_name`. I've chosen to filter data using the ID because the query is a tiny bit shorter that way.

Here's the output:

child_id	child_first_name	child_last_name	parent_first_name	parent_la
11	Sam Francis	Dijkstra	Mary	Dijkstra

It shows Sam Francis' mother is Mary Dijkstra, which is true.

Everything clear up till now? Good. Moving on!

Example 3: Get a Generation Number (or Tree Level) for Each Node

In this example, I want to list every person from the table and show which generation they belong to. What's the purpose of this? When I get that data, I can easily see who belongs to which generation: parents, children, grandchildren, etc.

I'll achieve that by using a CTE – not your everyday CTE, but a recursive CTE. If you need to refresh your CTE knowledge, here's an article explaining what a CTE is.

Here'is my query:

```
WITH RECURSIVE generation AS (
    SELECT id,
        first_name,
```

```
        last_name,
        parent_id,
        0 AS generation_number
    FROM parent_child
    WHERE parent_id IS NULL

UNION ALL

    SELECT child.id,
        child.first_name,
        child.last_name,
        child.parent_id,
        generation_number+1 AS generation_number
    FROM parent_child child
    JOIN generation g
      ON g.id = child.parent_id
)

SELECT first_name,
     last_name,
     generation_number
FROM generation;
```

As every recursive CTE, mine begins with two keywords: `WITH RECURSIVE`. I've named the CTE generation. In the first `SELECT` statement, I'm selecting IDs and names. Additionally, there's a new column called `generation_number` with a 0 for all the rows where `parent_id = NULL`. Why `NULL`? Because I know the person who is the predecessor of all other people has no parent in the table. Therefore, the value must be `NULL`.

I'm using `UNION ALL` to merge the result of this `SELECT` statement with the second one, which will be responsible for recursion. For `UNION ALL` to work, the number of columns and data types must be the same in both `SELECT` statements.

The recursive member again selects IDs and names. There's also the column `generation_number` with the value `generation_number+1`. With every recursion, 1 will be added to this column's previous value. As the query starts with 0, the first recursion will result in a 1 in the column `generation_number`, the second in a 2, and so on.

To make this all work, I've joined the table **`parent_child`** with the CTE itself where `id = parent_id`. The same principle applies as with self-joining tables: the table serves as data on children, the CTE serves as data on parents.

data on parents.

After writing the CTE, I need to use its data. I've done that by writing a simple `SELECT` statement that returns names and generation numbers from the CTE. Nice one, isn't it?

Here's how the result looks:

first_name	last_name	generation_number
Rosa	Wellington	0
Jon	Wellington	1
Joni	Wellington	1
Marge	Wellington	1
Mary	Dijkstra	2
Frank	Wellington	2
Jason	Wellington	2
Bobby	Wellington	2
Sammy	Wellington	2
Sarah	Wellington	2
Sam Francis	Dijkstra	3
Stephen	Wellington	3
Trent	Wellington	3
June	Wellington	3
Josephine	Wellington	3
Suzy	Wellington	3

With this result, I see that Rosa Wellington is the root node because her generation number is 0. All people with value 1 are her children, value 2 are grandchildren, and value 3 are great-grandchildren. If you check this in the source table, you'll find out everything I've said is true.

Learn how to process trees and graphs in SQL with our Recursive Queries course. The ultimate SQL challenge!

Example 4: List All Descendants

This example is an extension of the previous one. I want to show you how to list all descendants of a parent and show both parents' and

children's names.

This is the query:

```
WITH RECURSIVE generation AS (
    SELECT id,
         first_name,
         last_name,
         parent_id,
         0 AS generation_number
    FROM parent_child
    WHERE parent_id IS NULL

UNION ALL

    SELECT child.id,
         child.first_name,
         child.last_name,
         child.parent_id,
         generation_number+1 AS generation_number
    FROM parent_child child
    JOIN generation g
      ON g.id = child.parent_id

)

SELECT  g.first_name AS child_first_name,
        g.last_name AS child_last_name,
        g.generation_number,
        parent.first_name AS parent_first_name,
        parent.last_name AS parent_last_name
FROM generation g
JOIN parent_child parent
ON g.parent_id = parent.id
ORDER BY generation_number;
```

If you compare this query with the previous one, you'll see the CTE part is identical. No need for me to go through it again.

What is different is the `SELECT` statement referencing the CTE. But there are no new SQL concepts here either. The query selects the child's and parent's names and their generation number. I did this by again joining the CTE with the table **`parent_child`**. The CTE contains data for children, while the table contains data about parents. The last code line orders the result by the generation number.

The query returns exactly what I wanted:

The query returns exactly what I wanted.

child_first_name	child_last_name	generation_number	parent_first_nam
Marge	Wellington	1	Rosa
Joni	Wellington	1	Rosa
Jon	Wellington	1	Rosa
Frank	Wellington	2	Jon
Mary	Dijkstra	2	Jon
Jason	Wellington	2	Joni
Sarah	Wellington	2	Marge
Sammy	Wellington	2	Marge
Bobby	Wellington	2	Marge
Sam Francis	Dijkstra	3	Mary
Trent	Wellington	3	Frank
Stephen	Wellington	3	Frank
Suzy	Wellington	3	Sammy
Josephine	Wellington	3	Sammy
June	Wellington	3	Sammy

Or does it? Sure, it shows every child and their parent's name. But Rosa Wellington, the root node and matriarch of this family, is missing. And I didn't apply any filters to exclude her.

What happened? I actually did apply a filter by using `JOIN` in the last `SELECT` statement. Remember, `JOIN` returns only the matching rows from joined tables. Rosa Wellington is missing because she has no data about her parent; in her case, there's no data where `id` can match `parent_id`. If you want to include her too, use the `LEFT JOIN` in the last `SELECT`:

```
…
FROM generation g LEFT JOIN parent_child parent ON g.parent_id = parent.id
…
```

And the full result is here:

child_first_name	child_last_name	generation_number	parent_first_nam
Rosa	Wellington	0	NULL
Joni	Wellington	1	Rosa
Jon	Wellington	1	Rosa
Marge	Wellington	1	Rosa

child_first_name	child_last_name	generation_number	parent_first_nam
Mary	Dijkstra	2	Jon
Jason	Wellington	2	Joni
Sarah	Wellington	2	Marge
Sammy	Wellington	2	Marge
Bobby	Wellington	2	Marge
Frank	Wellington	2	Jon
Trent	Wellington	3	Frank
Stephen	Wellington	3	Frank
Suzy	Wellington	3	Sammy
Josephine	Wellington	3	Sammy
June	Wellington	3	Sammy
Sam Francis	Dijkstra	3	Mary

If you'd like to learn more about this complex query, here's an article dedicated to this example.

Example 5: Generate a Tree View of Hierarchical Data

The final example is the most complex one, but it's also the most fun. Or its output is, at least. It would be a shame to query tree structures without being able to show data in some kind of tree shape.

The task here is to show every person from the table. Also, every descendant has to be shown in a way that it's graphically obvious whose child they are and to which generation they belong. This is a tree view. I think it's best that you wait until I get to the query output to see what I mean by that.

Let's go to work! Again, the recursive CTE saves the day:

```
WITH RECURSIVE tree_view AS (

    SELECT id,
           parent_id,
           first_name,
           last_name,
           0 AS level,
           CAST(id AS varchar(50)) AS order_sequence
    FROM parent_child
    WHERE parent_id IS NULL

UNION ALL
```

```
    SELECT parent.id,
         parent.parent_id,
         parent.first_name,
         parent.last_name,
         level + 1 AS level,
         CAST(order_sequence || '_' || CAST(parent.id AS VARCHAR (50)) AS
VARCHAR(50)) AS order_sequence
    FROM parent_child parent
    JOIN tree_view tv
      ON parent.parent_id = tv.id
)

SELECT
   RIGHT('------------',level*3) || first_name || ' ' || last_name
     AS parent_child_tree
FROM tree_view
ORDER BY order_sequence;
```

You already know how recursive queries work. This time the CTE is named `tree_view`. The first `SELECT` statement selects some data from the table where `parent_id` is `NULL`. There's the column `level` with the value 0. And I've used the `CAST()` function to change the id data type to VARCHAR; you'll see why I need that.

We again use `UNION ALL` to merge two queries' results. The second `SELECT` statement again selects some data, with table **`parent_child`** joined with the CTE itself. The important thing is with every recursion, 1 will be added to the previous level. Also, the underscore and value from the column id will be added with every recursion. I need this little trick because I will use this column later to sort the output. That way, I'll show the tree view properly. To make sure you understand, here's one row from the table:

id	first_name	last_name	parent_id	order_sequence
1	Rosa	Wellington	NULL	1
2	Jon	Wellington	1	1_2
6	Frank	Wellington	2	1_2_6

The column value for Frank Wellington will be 1_2_6. Why? Because Rosa, as a first level, gets value 1. Jon Wellington is her son; his ID goes to the `order_sequence`, which now becomes 1_2. Then Frank's ID gets added and becomes 1_2_6. By doing this throughout the hierarchical structure, I get the column that I can use to show the output in the

desired way.

Back to the query. To get the result, you need a `SELECT` that uses the CTE data. I'm using the `RIGHT()` function here. It extracts a specified number of characters from the right. In my case, it removes the level*3 number of dashes for every level. I've also concatenated these dashes with the first and the last name. The result is sorted by the `order_sequence`.

Are you ready to see the tree view? Here it is:

parent_child_tree
Rosa Wellington
---Jon Wellington
------Mary Dijkstra
---------Sam Francis Dijkstra
------Frank Wellington
---------Stephen Wellington
---------Trent Wellington
---Joni Wellington
------Jason Wellington
---Marge Wellington
------Sarah Wellington
------Bobby Wellington
------Sammy Wellington
---------June Wellington
---------Josephine Wellington
---------Suzy Wellington

This simple graphical representation quite obviously shows the generational levels and who is who in this family tree. By the number of dashes, you can easily see that Jon, Joni, and Marge Wellington are Rosa's children. Mary Dijkstra, Frank, Jason, Sarah, Bobby, and Sammy Wellington are Rosa's grandchildren. It's also easy to see who their parents are. You can also see who the great-grandchildren are, but I'll leave that to you.

Before finishing this off, I'd also like to recommend this article on how querying tree structures work in Oracle.

querying tree structures work in Oracle.

Querying Parent-Child Trees Only Gets More Interesting

Parent-child tree structures are quite exciting. They are a completely different set of data to what you usually query in relational databases. I've shown you what these hierarchical structures are and how they are represented in a table.

Most importantly, I've shown you five queries that you can use to solve some of the most common problems regarding hierarchical data. As you saw, CTEs and recursive CTEs are vital to querying parent-child trees.

LearnSQL.com lets you learn SQL by writing SQL code on your own. You build your SQL skills gradually. Each new concept is reinforced by an interactive exercise. By actually writing SQL code, you build your confidence.

I'm sure you've already come across hierarchical data in your work. You probably realized you have to equip yourself with detailed recursive query knowledge to tackle such data. We have a Recursive Queries Course that will systematically lead you through CTEs in general, recursive queries, and how querying hierarchical data and graphs works in SQL.

Good luck with learning! And feel free to use all the queries I've shown you and to adapt them to your business needs.