**Hierarchical data in postgres**

[**POSTGRES**](https://coderwall.com/t/postgres/popular)

[**POSTGRESQL**](https://coderwall.com/t/postgresql/popular)

[**LTREE**](https://coderwall.com/t/ltree/popular)

This tip will try to answer the following questions:

* How can we represent a tree of data in postgres
* How can we efficiently query for any entire single node and all of it's children (and children's children).

**The test data**

Since we want to keep this simple we will assume our data is just a bunch of sections. A section just has a name and each section has a single parent section.

Section A

|--- Section A.1

Section B

|--- Section B.1

|--- Section B.1

|--- Section B.1.1

We'll use this simple data for examples below.

**Simple self-referencing**

When designing a self-referential table (something that joins itself to itself) the most obvious choice is to have some kind of parent\_id column on our table that references itself.

CREATE TABLE section (

id INTEGER PRIMARY KEY,

name TEXT,

parent\_id INTEGER REFERENCES section,

);

ALTER TABLE page ADD COLUMN parent\_id INTEGER REFERENCES page;

CREATE INDEX section\_parent\_id\_idx ON section (parent\_id);

Now insert our example data, using the parent\_id to related the nodes together:

INSERT INTO section (id, name, parent\_id) VALUES (1, 'Section A', NULL);

INSERT INTO section (id, name, parent\_id) VALUES (2, 'Section A.1', 1);

INSERT INTO section (id, name, parent\_id) VALUES (3, 'Section B', NULL);

INSERT INTO section (id, name, parent\_id) VALUES (4, 'Section B.1', 3);

INSERT INTO section (id, name, parent\_id) VALUES (5, 'Section B.2', 3);

INSERT INTO section (id, name, parent\_id) VALUES (6, 'Section B.2.1', 5);

This works great for simple queries such as, fetch the direct children of Section B:

SELECT \* FROM section WHERE parent = 3

but it will require complex or recursive queries for questions like fetch me all the children and children's children of Section B:

WITH RECURSIVE nodes(id,name,parent\_id) AS (

SELECT s1.id, s1.name, s1.parent\_id

FROM section s1 WHERE parent\_id = 3

UNION

SELECT s2.id, s2.name, s2.parent\_id

FROM section s2, nodes s1 WHERE s2.parent\_id = s1.id

)

SELECT \* FROM nodes;

So we have answered the "how to build a tree" part of the question, but are not happy with the "how to query for a node and all it's children" part.

Enter ltree. (Short for "label tree" - I think?).

**The ltree extension**

The [ltree extension](http://www.postgresql.org/docs/9.3/static/ltree.html) is a great choice for querying hierarchical data. This is especially true for self-referential relationships.

Lets rebuild the above example using ltree. We'll use the page's primary keys as the "labels" within our ltree paths and a special "root" label to denote the top of the tree.

CREATE EXTENSION ltree;

CREATE TABLE section (

id INTEGER PRIMARY KEY,

name TEXT,

parent\_path LTREE

);

CREATE INDEX section\_parent\_path\_idx ON section USING GIST (parent\_path);

We'll add in our data again, this time rather than using the id for the parent, we will construct an ltree path that represents the parent node.

INSERT INTO section (id, name, parent\_path) VALUES (1, 'Section 1', 'root');

INSERT INTO section (id, name, parent\_path) VALUES (2, 'Section 1.1', 'root.1');

INSERT INTO section (id, name, parent\_path) VALUES (3, 'Section 2', 'root');

INSERT INTO section (id, name, parent\_path) VALUES (4, 'Section 2.1', 'root.3');

INSERT INTO section (id, name, parent\_path) VALUES (4, 'Section 2.2', 'root.3');

INSERT INTO section (id, name, parent\_path) VALUES (5, 'Section 2.2.1', 'root.3.4');

Cool. So now we can make use of ltree's operators @> and <@ to answer our original question like:

SELECT \* FROM section WHERE parent\_path <@ 'root.3';

However we have introduced a few issues.

* Our simple parent\_id version ensured referential consistancy by making use of the REFERENCES constraint. We lost that by switching to ltree paths.
* Ensuring that the ltree paths are valid can be a bit of a pain, and if paths become stale for some reason your queries may return unexpected results or you may "orphan" nodes.

**The final solution**

To fix these issues we want a hybrid of our original parent\_id (for the referential consistency and simplicity of the child/parent relationship) and our ltree paths (for improved querying power/indexing). To achieve this we will hide the management of the ltree path behind a trigger and only ever update the parent\_idcolumn.

CREATE EXTENSION ltree;

CREATE TABLE section (

id INTEGER PRIMARY KEY,

name TEXT,

parent\_id INTEGER REFERENCES section,

parent\_path LTREE

);

CREATE INDEX section\_parent\_path\_idx ON section USING GIST (parent\_path);

CREATE INDEX section\_parent\_id\_idx ON section (parent\_id);

CREATE OR REPLACE FUNCTION update\_section\_parent\_path() RETURNS TRIGGER AS $$

DECLARE

path ltree;

BEGIN

IF NEW.parent\_id IS NULL THEN

NEW.parent\_path = 'root'::ltree;

ELSEIF TG\_OP = 'INSERT' OR OLD.parent\_id IS NULL OR OLD.parent\_id != NEW.parent\_id THEN

SELECT parent\_path || id::text FROM section WHERE id = NEW.parent\_id INTO path;

IF path IS NULL THEN

RAISE EXCEPTION 'Invalid parent\_id %', NEW.parent\_id;

END IF;

NEW.parent\_path = path;

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER parent\_path\_tgr

BEFORE INSERT OR UPDATE ON section

FOR EACH ROW EXECUTE PROCEDURE update\_section\_parent\_path();

Much better.

**More**

[Use json and plv8 to work with tree data](https://coderwall.com/p/z00-yw)