HypoPG Documentation

Julien Rouhaud

Dec 11, 2018

Contents:

HypoPG is a PostgreSQL extension, adding support for *Hypothetical objects*: *Hypothetical indexes* and *Hypothetical partitioning* (only for pg10 and above).

This extension is compatible with **PostgreSQL 9.2 and above**.

Note: This documentation is a work in progress. If you're looking for something and can't find it here, please report an issue so I can enhance the documentation.

CHAPTER 1

Hypothetical objects

HypoPG support two kinds of hypothetical objects: hypothetical indexes and hypothetical partitioning.

1.1 Hypothetical indexes

A hypothetical, or virtual, index is an index that doesn't really exists, and thus doesn't cost CPU, disk or any resource to create. They're useful to know if specific indexes can increase performance for problematic queries, since you can know if PostgreSQL will use these indexes or not without having to spend resources to create them.

1.2 Hypothetical partitioning

Hypothetical partitioning, available for PostgreSQL servers version 10 and above, is a real table on which you can hypothetically apply partitining scheme as you would do for declarative partitioning. PostgreSQL will act as if this table was really partitioned, so you can know quickly test multiple partitioning scheme, and you can see how each of them will change your queries behavior and check which one is the best for your specific appliation.

CHAPTER 2

Installation

2.1 Requirements

- PostgreSQL 9.2+

2.2 Packages

Hypopg is available as a package on some GNU/Linux distributions:

- RHEL/Centos

  HypoPG is available as a package using the PGDG packages.

  Once the PGDG repository is setup, you just need to install the package. As root:

  ```
  yum install hypopg
  ```

- Archlinux

  Hypopg is available on the AUR repository.

  If you have **yaourt** setup, you can simply install the *hypopg-git* package with the following command:

  ```
  yaourt -S hypopg-git
  ```

  Otherwise, look at the official documentation to manually install the package.

Note: Installing this package will use the current development version. If you want to install a specific version, please see the *Installation from sources* section.

2.3 Installation from sources

To install HypoPG from sources, you need the following extra requirements:

- PostgreSQL development packages

Note: On Debian/Ubuntu systems, the development packages are named *postgresql-server-dev-X*, X being the major version.

On RHEL/Centos systems, the development packages are named *postgresqlX-devel*, X being the major version.

- A C compiler and *make*
- *unzip*
- optionally the *wget* tool
- a user with *sudo* privilege, or a root access

Note: If you don't have *sudo* or if you user isn't authorized to issue command as root, you should do all the following commands as **root**.

First, you need to download HypoPG source code. If you want the development version, you can download it from here, or via command line:

```
wget https://github.com/HypoPG/hypopg/archive/master.zip
```

If you want a specific version, you can chose the version you want here and follow the related download link. For instance, if you want to install the version 1.0.0, you can download it from the command line with the following command:

```
wget https://github.com/HypoPG/hypopg/archive/1.0.0.zip
```

Then, you need to extract the downloaded archive with *unzip* and go to the extracted directory. For instance, if you downloaded the latest development version:

```
unzip master.zip
cd hypopg-master
```

You can now compile and install HypoPG. Simply run:

```
make
sudo make install
```

Note: If you were doing these commands as **root**, you don't need to use sudo. The last command should therefore be:

```
make install
```

If no errors occured, HypoPG is now available! If you need help on how to use it, please refer to the *Usage* section.

CHAPTER 3

Usage

3.1 Introduction

HypoPG is useful if you want to check if some index would help one or multiple queries. Therefore, you should already know what are the queries you need to optimize, and ideas on which indexes you want to try.

Also, the hypothetical indexes that HypoPG will create are not stored in any catalog, but in your connection private memory. Therefore, it won't bloat any table and won't impact any concurrent connection.

Also, since the hypothetical indexes doesn't really exists, HypoPG makes sure they will only be used using a simple EXPLAIN statement (without the ANALYZE option).

3.2 Install the extension

As any other extension, you have to install it on all the databases where you want to be able to use it. This is simply done executing the following query, connected on the database you want to install HypoPG with a user having enough privileges:

```
CREATE EXTENSION hypopg ;
```

HypoPG is now available. You can check easily if the extension is present using psql:

```
\dx
                    List of installed extensions
  Name   | Version |   Schema   |              Description
---------+---------+------------+-------------------------------------
 hypopg  | 1.1.0   | public     | Hypothetical indexes for PostgreSQL
 plpgsql | 1.0     | pg_catalog | PL/pgSQL procedural language
(2 rows)
```

As you can see, hypopg version 1.1.0 is installed. If you need to check using plain SQL, please refer to the pg_extension table documentation.

3.3 Create a hypothetical index

Note: Using HypoPG require some knowledge on the **EXPLAIN** command. If you need more information about this command, you can check the official documentation. There are also a lot of very good resources available.

For clarity, let's see how it works with a very simple test case:

```
CREATE TABLE hypo (id integer, line text) ;
INSERT INTO hypo SELECT i, 'line ' || i FROM generate_series(1, 100000) i ;
VACUUM ANALYZE hypo ;
```

This table doesn't have any index. Let's assume we want to check if an index would help a simple query. First, let's see how it behaves:

```
EXPLAIN SELECT val FROM hypo WHERE id = 1;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on hypo  (cost=0.00..1791.00 rows=1 width=14)
   Filter: (id = 1)
(2 rows)
```

A plain sequential scan is used, since no index exists on the table. A simple btree index on the **id** column should help this query. Let's check with HypoPG. The function **hypopg_create_index()** will accept any standard **CREATE INDEX** statement(s) (any other statement passed to this function will be ignored), and create a hypothetical index for each:

```
SELECT * FROM hypopg_create_index('CREATE INDEX ON hypo (id)') ;
 indexrelid |      indexname
------------+----------------------
      18284 | <18284>btree_hypo_id
(1 row)
```

The function returns two columns:

- the object identifier of the hypothetical index
- the generated hypothetical index name

We can run the EXPLAIN again to see if PostgreSQL would use this index:

```
EXPLAIN SELECT val FROM hypo WHERE id = 1;
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Index Scan using <18284>btree_hypo_id on hypo  (cost=0.04..8.06 rows=1 width=10)
   Index Cond: (id = 1)
(2 rows)
```

Yes, PostgreSQL would use such an index. Just to be sure, let's check that the hypothetical index won't be used to acually run the query:

```
EXPLAIN ANALYZE SELECT val FROM hypo WHERE id = 1;
                                              QUERY PLAN
-----------------------------------------------------------------------------------------------
↪-------------
 Seq Scan on hypo  (cost=0.00..1791.00 rows=1 width=10) (actual time=0.046..46.390␣
↪rows=1 loops=1)
```

(continues on next page)

(continued from previous page)

```
   Filter: (id = 1)
   Rows Removed by Filter: 99999
 Planning time: 0.160 ms
 Execution time: 46.460 ms
(5 rows)
```

That's all you need to create hypothetical indexes and see if PostgreSQL would use such indexes.

3.4 Manipulate hypothetical indexes

Some other convenience functions are available:

- **hypopg_list_indexes()**: list all hypothetical indexes that have been created

```
SELECT * FROM hypopg_list_indexes()
 indexrelid |      indexname       | nspname | relname | amname
------------+----------------------+---------+---------+--------
      18284 | <18284>btree_hypo_id | public  | hypo    | btree
(1 row)
```

- **hypopg_get_indexdef(oid)**: get the CREATE INDEX statement that would recreate a stored hypothetical index

```
SELECT indexname, hypopg_get_indexdef(indexrelid) FROM hypopg_list_indexes() ;
      indexname       |             hypopg_get_indexdef
----------------------+---------------------------------------------
 <18284>btree_hypo_id | CREATE INDEX ON public.hypo USING btree (id)
(1 row)
```

- **hypopg_relation_size(oid)**: estimate how big a hypothetical index would be:

```
SELECT indexname, pg_size_pretty(hypopg_relation_size(indexrelid))
  FROM hypopg_list_indexes() ;
      indexname       | pg_size_pretty
----------------------+----------------
 <18284>btree_hypo_id | 2544 kB
(1 row)
```

- **hypopg_drop_index(oid)**: remove the given hypothetical index
- **hypopg_reset()**: remove all hypothetical indexes

3.5 Hypothetical partitioning

Note: This is only possible for PostgreSQL 10 an above. The partitioning possibilites depend on the PostgreSQL version. For instance, you can't create a hypothetical hash partition on using PostgreSQL 10.

For clarity, let's see how it works with a very simple test case:

```
CREATE TABLE hypo_part_range (id integer, val text);
INSERT INTO hypo_part_range SELECT i, 'line ' || i FROM generate_series(1, 29999) i;
```

This is a simple table, containing some rows and without indexes. Trying to retrieve a row will do as expected:

```
EXPLAIN SELECT * FROM hypo_part_range WHERE id = 2;
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on hypo_part_range  (cost=0.00..537.99 rows=1 width=14)
   Filter: (id = 2)
(2 rows)
```

Now, let's try to hypothetically partition this table with a range partitioning scheme. For that, we have two functions:

- **hypopg_partition_table**: it has two mandatory arguments. The first argument is the table to be hypothetically partitioned, and the second is the *PARTITION BY* clause, as you would use for declarative partitioning
- **hypopg_add_partition**: it has two mandatory arguments, and one optional. The first mandatory argument is the partitiong name, the second is the *PARTITION OF* clause, and the optional argument is a *PARTITION BY* clause, if you want to declare multiple level of partitioning.

For instance:

```
SELECT hypopg_partition_table('hypo_part_range', 'PARTITION BY RANGE(id)');
SELECT tablename FROM hypopg_add_partition('hypo_part_range_1_10000', 'PARTITION OF␣
↪hypo_part_range FOR VALUES FROM (1) TO (10000)');
SELECT tablename FROM hypopg_add_partition('hypo_part_range_10000_20000', 'PARTITION␣
↪OF hypo_part_range FOR VALUES FROM (10000) TO (20000)');
SELECT tablename FROM hypopg_add_partition('hypo_part_range_20000_30000', 'PARTITION␣
↪OF hypo_part_range FOR VALUES FROM (20000) TO (30000)');
```

Note: If you need to declare bounds on a textual column, the dollar-quoting notation will be helpful. For instance:

```
SELECT hypopg_add_partition('p_name', $$PARTITION OF tbl FOR VALUES FROM 'aaa' TO 'aab
↪'$$);
```

Now, let's see what happens if we try to retrieve a row of the hypothetically partitioned table:

```
EXPLAIN SELECT * FROM hypo_part_range WHERE id = 2;
                                          QUERY PLAN
-----------------------------------------------------------------------------------------
↪----------
 Append  (cost=0.00..179.95 rows=1 width=14)
   ->  Seq Scan on hypo_part_range hypo_part_range_1_10000  (cost=0.00..179.95 rows=1␣
↪width=14)
```

We can see that since there's an Append node, PostgreSQL acted as if the table was partitioned, and that all but one partition was pruned.

It's also possible to create a hypothetical index on the hypothetical partitions:

```
SELECT hypopg_create_index('CREATE INDEX on hypo_part_range_1_10000 (id)');
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------
↪-----------------------------------------------------
 Append  (cost=0.04..8.06 rows=1 width=14)
   ->  Index Scan using <258199>btree_hypo_part_range_1_10000_id on hypo_part_range␣
↪hypo_part_range_1_10000  (cost=0.04..8.05 rows=1 width=14)
         Index Cond: (id = 2)
(3 rows)
```

3.6 Manipulate hypothetical partitions

Some other convenience functions are available:

- **hypopg_table()**: list all hypothetical partitions that have been created
- **hypopg_analyze(regclass, fraction)**: perform an operation similar to ANALYZE on a hypothetically partitioned table, to get better estimates
- **hypopg_statistic()**: returns the list of statistics gathered by previous runs of **hypopg_analyze**, in the same format as *pg_statistic*. For an easier reading, the view **hypopg_stats** exists, which returns the data in the same format as *pg_stats*
- **hypopg_drop_table(oid)**: delete a previously created partition, or unpartition a hypothetically partitioned table (including the stored statistics if any)
- **hypopg_reset_table()**: remove all previously created hypothetical partition (inclufing the stored statistics if any)

CHAPTER 4

Contributing

HypoPG is an open source project, distributed under the PostgreSQL licence.

4.1 Talk

If you have suggestions, feature request or just want to say hi you can join the **#hypopg** IRC channel on freenode.

4.2 Bug reports

If you've found a bug, please report it on the HypoPG bug-tracker on Github.

4.3 Hacking

If you want to fix a bug, enhance the documentation or develop new features, feel free to clone the git repository on Github.