PG Hint_Plan

pg_hint_plan documentation

Introduction

Markdown format is kept as a main format, relying on python sphinx and myst_parser to render an HTML documentation if needed.

Note that while markdown is more readable as raw text, it's a way simpler syntax that lacks a lot of features that reStructuredText offers. Relying on sphinx gives us an opportunity to later write parts of the documentation in reStructuredText if needed, but also offers other appealing features like multilingual documentation.

Readthedocs is the expected target, so use its theme and follow its recommendation about pinning various requirement versions.

Building the doc locally

The documentation can be built locally easily using

```
make -C docs/ html
```

The rendered documentation will be generated in docs/html/_build/html

Note that you need to have all python prerequirements installed, which can be done using:

```
pip install -r docs/requirements.txt
```

If you need to update the requirements (which shouldn't be needed frequently), update the docs/requirements.in and generate the target docs/requirements.txt using pip-compile. See the link about this tool below for more details on how to use it.

Translation

Note that each translator has to follow all those steps whenever the translation needs to be updated. Note also that those commands assume that the current working directory is docs/.

- Bootstrapping the translation (the .pot files) is simply done using

```
make gettext
```

This will generate the various .pot file in _build/gettext.

- The per-language translation files (the .po files) can then be generated. We currently only support Japanese, the rest of the commands will assume a single Japanese translation. Those files can be generated using:

sphinx-intl update -p _build/gettext -l ja

The files are generated (or updated) in the docs/locale/ja/LC_MESSAGES/.

- You can then translate the .po file with any editor (poedit, vim…)
- The translated documentation can be built using:

make -e SPHINXOPTS="-D language='ja' " html

- If everything is ok, you can commit the modifications in the .po files.

References

References if you're interested in the various design choices:

- quickstart for RTD with sphinx: https://docs.readthedocs.io/en/stable/intro/getting-started-with-sphinx.html
- reproducible builds: https://docs.readthedocs.io/en/stable/guides/reproducible-builds.html
- myst parser: https://myst-parser.readthedocs.io
- pip-tools / pip-compile: https://pip-tools.readthedocs.io
- RTD sphinx theme: https://sphinx-rtd-theme.readthedocs.io
- Internationalization: https://www.sphinx-doc.org/en/master/usage/advanced/intl.html https://docs.readthedocs.io/en/stable/localization.html#projects-with-multiple-translations-sphinx-only # Errors

`pg_hint_plan` stops hint parsing on any error and will uses the hints already parsed. Here are some typical errors.

Syntax errors

Any syntactical errors or wrong hint names are reported as a syntax error. These errors are reported in the server log with the message level specified by `pg_hint_plan.message_level` if `pg_hint_plan.debug_print` is on and above.

Incorrect Object definitions

Incorrect object definitions result in silently ignoring the hints. This kind of error is reported as a "Not Used Hint" in the server logs.

Redundant or conflicting hints

The last hint is considered when redundant hints are defined or hints conflict with each other. This kind of error is reported as a duplicated hints.

Nested comments

Hint comments cannot be recursive. If detected, hint parsing is immediately stopped and all the hints already parsed are ignored. (functional-limitations)=

Functional limitations

Influence of planner GUC parameters

The planner does not try to consider joining order for FROM clause entries more than `from_collapse_limit`. `pg_hint_plan` cannot affect the joining order in this case.

Hints trying to enforce non-executable plans

Planner chooses any executable plans when the enforced plan cannot be executed:

- `FULL OUTER JOIN` to use nested loop.
- Use of indexes that do not have columns used in quals.
- TID scans for queries without ctid conditions.

Queries in ECPG

ECPG removes comments in queries written as embedded SQLs so hints cannot be passed to it. The only exception `EXECUTE`, that passes the query string to the server as-is. The hint table can be used in the case.

Query Identifiers

When `compute_query_id` is enabled, PostgreSQL generates a query ID, ignoring comments. Hence, queries with different hints, still written the same way, may compute the same query ID. # Details in hinting

Syntax and placement

`pg_hint_plan` reads hints from only the first block comment and stops parsing from any characters except alphabetical characters, digits, spaces, underscores, commas and parentheses. In the following example, `HashJoin(a b)` and `SeqScan(a)` are parsed as hints, but `IndexScan(a)` and `MergeJoin(a b)` are not:

```
=# /*+
     HashJoin(a b)
     SeqScan(a)
    */
   /*+ IndexScan(a) */
   EXPLAIN SELECT /*+ MergeJoin(a b) */ *
     FROM pgbench_branches b
     JOIN pgbench_accounts a ON b.bid = a.bid
     ORDER BY a.aid;
                                  QUERY PLAN
---------------------------------------------------------------------------------------
 Sort  (cost=31465.84..31715.84 rows=100000 width=197)
```

```
   Sort Key: a.aid
   ->  Hash Join  (cost=1.02..4016.02 rows=100000 width=197)
         Hash Cond: (a.bid = b.bid)
         ->  Seq Scan on pgbench_accounts a  (cost=0.00..2640.00 rows=100000 width=97)
         ->  Hash  (cost=1.01..1.01 rows=1 width=100)
               ->  Seq Scan on pgbench_branches b  (cost=0.00..1.01 rows=1 width=100)
(7 rows)
```

Using with PL/pgSQL

`pg_hint_plan` works for queries in PL/pgSQL scripts with some restrictions.

- Hints affect only on the following kind of queries:
  - Queries that return one row (`SELECT`, `INSERT`, `UPDATE` and `DELETE`)
  - Queries that return multiple rows (`RETURN QUERY`)
  - Dynamic SQL statements (`EXECUTE`)
  - Cursor open (`OPEN`)
  - Loop over result of a query (`FOR`)
- A hint comment has to be placed after the first word in a query as preceding comments are not sent as a part of this query.

```
=# CREATE FUNCTION hints_func(integer) RETURNS integer AS $$
   DECLARE
     id  integer;
     cnt integer;
   BEGIN
     SELECT /*+ NoIndexScan(a) */ aid
       INTO id FROM pgbench_accounts a WHERE aid = $1;
     SELECT /*+ SeqScan(a) */ count(*)
       INTO cnt FROM pgbench_accounts a;
     RETURN id + cnt;
   END;
   $$ LANGUAGE plpgsql;
```

Upper and lower case handling in object names

Unlike the way PostgreSQL handles object names, `pg_hint_plan` compares bare object names in hints against the database internal object names in a case-sensitive manner. Therefore, an object name TBL in a hint matches only “TBL” in the database and does not match any unquoted names like TBL, tbl or Tbl.

Escaping special characters in object names

The objects defined in a hint’s parameter can use double quotes if they include parentheses, double quotes and white spaces. The escaping rules are the same as PostgreSQL.

Distinction between multiple occurences of a table

pg_hint_plan identifies the target object by using aliases if any. This behavior is useful to point to a specific occurrence among multiple occurrences of one table.

```
=# /*+ HashJoin(t1 t1) */
   EXPLAIN SELECT * FROM s1.t1
     JOIN public.t1 ON (s1.t1.id=public.t1.id);
INFO:  hint syntax error at or near "HashJoin(t1 t1)"
DETAIL:  Relation name "t1" is ambiguous.
...
=# /*+ HashJoin(pt st) */
   EXPLAIN SELECT * FROM s1.t1 st
     JOIN public.t1 pt ON (st.id=pt.id);
                             QUERY PLAN
---------------------------------------------------------------------
 Hash Join  (cost=64.00..1112.00 rows=28800 width=8)
   Hash Cond: (st.id = pt.id)
   ->  Seq Scan on t1 st  (cost=0.00..34.00 rows=2400 width=4)
   ->  Hash  (cost=34.00..34.00 rows=2400 width=4)
         ->  Seq Scan on t1 pt  (cost=0.00..34.00 rows=2400 width=4)
```

Underlying tables of views or rules

Hints are not applicable on views, but they can affect the queries within the view if the object names match the names in the expanded query on the view. Assigning aliases to the tables in a view enables them to be manipulated from outside the view.

```
=# CREATE VIEW v1 AS SELECT * FROM t2;
=# EXPLAIN /*+ HashJoin(t1 v1) */
          SELECT * FROM t1 JOIN v1 ON (c1.a = v1.a);
                            QUERY PLAN
------------------------------------------------------------------
 Hash Join  (cost=3.27..18181.67 rows=101 width=8)
   Hash Cond: (t1.a = t2.a)
   ->  Seq Scan on t1  (cost=0.00..14427.01 rows=1000101 width=4)
   ->  Hash  (cost=2.01..2.01 rows=101 width=4)
         ->  Seq Scan on t2  (cost=0.00..2.01 rows=101 width=4)
```

Inheritance

Hints can only point to the parent of an inheritance tree and the hints affect all the tables in an inheritance tree. Hints pointing directly to inherited children have no effect.

Hints in multistatements

One multistatement can have exactly one hint comment and the hint affects all of the individual statements in the multistatement.

VALUES expressions

VALUES expressions in FROM clause are named as *VALUES* internally these can be hinted if it is the only VALUES of a query. Two or more VALUES expressions in a query cannot be distinguished by looking at an EXPLAIN result, resulting in ambiguous results:

```
=# /*+ MergeJoin(*VALUES*_1 *VALUES*) */
   EXPLAIN SELECT * FROM (VALUES (1, 1), (2, 2)) v (a, b)
     JOIN (VALUES (1, 5), (2, 8), (3, 4)) w (a, c) ON v.a = w.a;
INFO:  pg_hint_plan: hint syntax error at or near "MergeJoin(*VALUES*_1 *VALUES*) "
DETAIL:  Relation name "*VALUES*" is ambiguous.
                               QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=0.05..0.12 rows=2 width=16)
   Hash Cond: ("*VALUES*_1".column1 = "*VALUES*".column1)
   ->  Values Scan on "*VALUES*_1"  (cost=0.00..0.04 rows=3 width=8)
   ->  Hash  (cost=0.03..0.03 rows=2 width=8)
         ->  Values Scan on "*VALUES*"  (cost=0.00..0.03 rows=2 width=8)
```

Subqueries

Subqueries context can be occasionally hinted using the name ANY_subquery:

```
IN (SELECT ... {LIMIT | OFFSET ...} ...)
= ANY (SELECT ... {LIMIT | OFFSET ...} ...)
= SOME (SELECT ... {LIMIT | OFFSET ...} ...)
```

For these syntaxes, the planner internally assigns the name to the subquery when planning joins on tables including it, so join hints are applicable on such joins using the implicit name. For example:

```
=# /*+HashJoin(a1 ANY_subquery)*/
   EXPLAIN SELECT *
     FROM pgbench_accounts a1
   WHERE aid IN (SELECT bid FROM pgbench_accounts a2 LIMIT 10);
                                         QUERY PLAN

---------------------------------------------------------------------------------------
 Hash Semi Join  (cost=0.49..2903.00 rows=1 width=97)
   Hash Cond: (a1.aid = a2.bid)
   ->  Seq Scan on pgbench_accounts a1  (cost=0.00..2640.00 rows=100000 width=97)
   ->  Hash  (cost=0.36..0.36 rows=10 width=4)
```

```
         ->  Limit  (cost=0.00..0.26 rows=10 width=4)
               ->  Seq Scan on pgbench_accounts a2  (cost=0.00..2640.00 rows=100000 width=4)
```

Using `IndexOnlyScan` hint

Index scan may be unexpectedly performed on another index when the index specified in IndexOnlyScan hint cannot perform an index only scan.

About `NoIndexScan`

A `NoIndexScan` hint implies `NoIndexOnlyScan`.

Parallel hints and `UNION`

A `UNION` can run in parallel only when all underlying subqueries are parallel-safe. Hence, enforcing parallel on any of the subqueries will let a parallel-executable `UNION` run in parallel. Meanwhile, a parallel hint with zero workers prevents a scan from being executed in parallel.

Setting `pg_hint_plan` parameters by Set hints

`pg_hint_plan` parameters influence their own behavior so some parameters will not work as one could expect:

- Hints to change `enable_hint`, `enable_hint_table` are ignored even though they are reported as "used hints" in debug logs.
- Setting `debug_print` and `message_level` in the middle of query processing. (hint-list)= # Hint list

The available hints are listed below.

Group	Format	Description
Scan method	`SeqScan(table)`	Forces sequential scan on the table.
	`TidScan(table)`	Forces TID scan on the table.
	`IndexScan(table[ index...])`	Forces index scan on the table. Restricts to specified indexes if any.
	`IndexOnlyScan(table[ index...])`	Forces index-only scan on the table. Restricts to specified indexes if any. Index scan may be used if index-only scan is not available.
	`BitmapScan(table[ index...])`	Forces bitmap scan on the table. Restricts to specified indexes if any.

Group	Format	Description
	`IndexScanRegexp(table[ POSIX Regexp...])IndexOnlyScanRegexp(table[ POSIX Regexp...])BitmapScanRegexp(table[ POSIX Regexp...])`	Forces index scan, index-only scan (For PostgreSQL 9.2 and later) or bitmap scan on the table. Restricts to indexes that matches the specified POSIX regular expression pattern.
	`NoSeqScan(table)`	Forces to *not* do sequential scan on the table.
	`NoTidScan(table)`	Forces to *not* do TID scan on the table.
	`NoIndexScan(table)`	Forces to *not* do index scan and index-only scan on the table.
	`NoIndexOnlyScan(table)`	Forces to *not* do index only scan on the table.
	`NoBitmapScan(table)`	Forces to *not* do bitmap scan on the table.
Join method	`NestLoop(table table[ table...])`	Forces nested loop for the joins on the tables specified.
	`HashJoin(table table[ table...])`	Forces hash join for the joins on the tables specified.
	`MergeJoin(table table[ table...])`	Forces merge join for the joins on the tables specified.
	`NoNestLoop(table table[ table...])`	Forces to *not* do nested loop for the joins on the tables specified.
	`NoHashJoin(table table[ table...])`	Forces to *not* do hash join for the joins on the tables specified.
	`NoMergeJoin(table table[ table...])`	Forces to *not* do merge join for the joins on the tables specified.
Join order	`Leading(table table[ table...])`	Forces join order as specified.
	`Leading(<join pair>)`	Forces join order and directions as specified. A join pair is a pair of tables and/or other join pairs enclosed by parentheses, which can make a nested structure.
Behavior control on Join	`Memoize(table table[ table...])`	Allows the topmost join of a join among the specified tables to Memoize the inner result. Not enforced.
	`NoMemoize(table table[ table...])`	Inhibits the topmost join of a join among the specified tables from Memoizing the inner result.

Group	Format	Description
Row number correction	`Rows(table table[ table...] correction)`	Corrects row number of a result of the joins on the tables specified. The available correction methods are absolute (#), addition (+), subtract (-) and multiplication (*). should be a string that strtod() can understand.
Parallel query configuration	`Parallel(table <# of workers> [soft|hard])`	Enforces or inhibits parallel execution of the specified table. <# of workers> is the desired number of parallel workers, where zero means inhibiting parallel execution. If the third parameter is soft (default), it just changes max_parallel_workers_per_gather and leaves everything else to the planner. Hard enforces the specified number of workers.
GUC	`Set(GUC-param value)`	Sets GUC parameter to the value defined while planner is running.

The hint table

Hints can be specified in a comment, still this can be inconvenient in the case where queries cannot be edited. In the case, hints can be placed in a special table named `"hint_plan.hints"`. The table consists of the following columns:

column	description
`id`	Unique number to identify a row for a hint. This column is filled automatically by sequence.
`query_id`	A unique query ID, generated by the backend when the GUC compute_query_id is enabled
`application_name`	The value of `application_name` where sessions can apply a hint. The hint in the example below applies to sessions connected from psql. An empty string implies that all sessions will apply the hint.
`hints`	Hint phrase. This must be a series of hints excluding surrounding comment marks.

The following example shows how to operate with the hint table.

```
=# EXPLAIN (VERBOSE, COSTS false) SELECT * FROM t1 WHERE t1.id = 1;
               QUERY PLAN
----------------------------------------
 Seq Scan on public.t1
   Output: id, id2
   Filter: (t1.id = 1)
 Query Identifier: -7164653396197960701
(4 rows)
=# INSERT INTO hint_plan.hints(query_id, application_name, hints)
     VALUES (-7164653396197960701, '', 'SeqScan(t1)');
INSERT 0 1
=# UPDATE hint_plan.hints
     SET hints = 'IndexScan(t1)'
     WHERE id = 1;
UPDATE 1
=# DELETE FROM hint_plan.hints WHERE id = 1;
DELETE 1
```

The hint table is owned by the extension owner and has the same default privileges as of the time of its creation, during `CREATE EXTENSION`. Hints in the hint table are prioritized over hints in comments.

The query ID can be retrieved with `pg_stat_statements` or with `EXPLAIN (VERBOSE)`.

Types of hints

Hinting phrases are classified in multiple types based on what kind of object and how they can affect the planner. See Hint list for more details.

Hints for Scan methods

Scan method hints enforce specific scanning methods on the target table. `pg_hint_plan` recognizes the target table by alias names if any. These are for example `SeqScan` or `IndexScan`.

Scan hints work on ordinary tables, inheritance tables, UNLOGGED tables, temporary tables and system catalogs. External (foreign) tables, table functions, VALUES clause, CTEs, views and subqueries are not affected.

```
=# /*+
     SeqScan(t1)
     IndexScan(t2 t2_pkey)
    */
   SELECT * FROM table1 t1 JOIN table table2 t2 ON (t1.key = t2.key);
```

Hints for Join methods

Join method hints enforce the join methods of the joins involving the specified tables.

This can affect joins only on ordinary tables. Inheritance tables, UNLOGGED tables, temporary tables, external (foreign) tables, system catalogs, table functions, VALUES command results and CTEs are allowed to be in the parameter list. Joins on views and subqueries are not affected.

Hints for Joining order

This hint, named "Leading", enforces the order of join on two or more tables. There are two methods of enforcing it. The first method enforces a specific order of joining but does not restrict the direction at each join level. The second method enforces the join direction additionally. See hint list for more details. For example:

```
=# /*+
     NestLoop(t1 t2)
     MergeJoin(t1 t2 t3)
     Leading(t1 t2 t3)
    */
   SELECT * FROM table1 t1
     JOIN table table2 t2 ON (t1.key = t2.key)
     JOIN table table3 t3 ON (t2.key = t3.key);
```

Hints for Row number corrections

This hint, named "Rows", changes the row number estimation of joins that comes from restrictions in the planner. For example:

```
=# /*+ Rows(a b #10) */ SELECT... ; Sets rows of join result to 10
=# /*+ Rows(a b +10) */ SELECT... ; Increments row number by 10
=# /*+ Rows(a b -10) */ SELECT... ; Subtracts 10 from the row number.
=# /*+ Rows(a b *10) */ SELECT... ; Makes the number 10 times larger.
```

Hints for parallel plans

This hint, named `Parallel`, enforces parallel execution configuration on scans. The third parameter specifies the strength of the enforcement. `soft` means that `pg_hint_plan` only changes `max_parallel_worker_per_gather` and leaves all the others to the planner to set. `hard` changes other planner parameters so as to forcibly apply the update. This can affect ordinary tables, inheritance parents, unlogged tables and system catalogs. External tables, table functions, `VALUES` clauses, CTEs, views and subqueries are not affected. Internal tables of a view can be specified by its real name or its alias as the target object. The following example shows that the query is enforced differently on each table:

```
=# EXPLAIN /*+ Parallel(c1 3 hard) Parallel(c2 5 hard) */
   SELECT c2.a FROM c1 JOIN c2 ON (c1.a = c2.a);
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=2.86..11406.38 rows=101 width=4)
   Hash Cond: (c1.a = c2.a)
   ->  Gather  (cost=0.00..7652.13 rows=1000101 width=4)
         Workers Planned: 3
         ->  Parallel Seq Scan on c1  (cost=0.00..7652.13 rows=322613 width=4)
   ->  Hash  (cost=1.59..1.59 rows=101 width=4)
         ->  Gather  (cost=0.00..1.59 rows=101 width=4)
               Workers Planned: 5
               ->  Parallel Seq Scan on c2  (cost=0.00..1.59 rows=59 width=4)

=# EXPLAIN /*+ Parallel(tl 5 hard) */ SELECT sum(a) FROM tl;
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Finalize Aggregate  (cost=693.02..693.03 rows=1 width=8)
   ->  Gather  (cost=693.00..693.01 rows=5 width=8)
         Workers Planned: 5
         ->  Partial Aggregate  (cost=693.00..693.01 rows=1 width=8)
               ->  Parallel Seq Scan on tl  (cost=0.00..643.00 rows=20000 width=4)
```

GUC parameters set during planning

`Set` hints change GUC parameters just while planning. GUC parameter shown in Query Planning can have the expected effects on planning unless an other hint conflicts with the planner method configuration parameters. When multiple hints change the same GUC, the last hint takes effect. GUC parameters for `pg_hint_plan` are also settable by this hint but it may not work as expected. See Functional limitations for details.

```
=# /*+ Set(random_page_cost 2.0) */
   SELECT * FROM table1 t1 WHERE key = 'value';
...
```

(guc-parameters-for-pg__hint__plan)= ## GUC parameters for `pg_hint_plan`

The following GUC parameters affect the behavior of `pg_hint_plan`:

Parameter name	Description	Default
`pg_hint_plan.enable_hint`	True enables `pg_hint_plan`.	`on`
`pg_hint_plan.enable_hint_table`	True enables hinting by table.	`off`

Parameter name	Description	Default
`pg_hint_plan.parse_messages`	Specifies the log level of hint parse error. Valid values are `error`, `warning`, `notice`, `info`, `log`, `debug`.	`INFO`
`pg_hint_plan.debug_print`	Controls debug print and verbosity. Valid values are `off`, `on`, `detailed` and `verbose`.	`off`
`pg_hint_plan.message_level`	Specifies message level of debug print. Valid values are `error`, `warning`, `notice`, `info`, `log`, `debug`.	`INFO`

Installation

This section describes the installation steps.

Building binary module

Simply run `make` at the top of the source tree, then `make install` as an appropriate user. The `PATH` environment variable should be set properly to point to a PostgreSQL set of binaries:

```
$ tar xzvf pg_hint_plan-1.x.x.tar.gz
$ cd pg_hint_plan-1.x.x
$ make
$ su
$ make install
```

Installing from a binary package

On Debian and Ubuntu `pg_hint_plan` is available as a binary package from the pgdg (PostgreSQL Global Development Group) repository. Assuming you've already added the repository to `apt` sources, installing the package is as simple as:

```
sudo apt install postgresql-<postgres version>-pg-hint-plan
```

Please visit https://www.postgresql.org/download/linux/ if you need help at adding the repository.

Loading `pg_hint_plan`

`pg_hint_plan` does not require `CREATE EXTENSION`. Loading it with a `LOAD` command will activate it and of course you can load it globally by setting `shared_preload_libraries` in `postgresql.conf`. Or you might be interested in `ALTER USER SET`/`ALTER DATABASE SET` for automatic loading in specific sessions.

```
postgres=# LOAD 'pg_hint_plan';
LOAD
```

Run `CREATE EXTENSION` and `SET pg_hint_plan.enable_hint_table TO on` if you are planning to use the hint table. # Requirements

pg_hint_plan 1.7 requires PostgreSQL 17.

PostgreSQL versions tested

- Version 17

OS versions tested

- CentOS 8.5

See also

References

- EXPLAIN
- SET
- Server Config
- Parallel Plans # Synopsis

`pg_hint_plan` makes it possible to tweak PostgreSQL execution plans using "hints" in SQL comments, as of `/*+ SeqScan(a) */`.

PostgreSQL uses a cost-based optimizer, which utilizes data statistics, not static rules. The planner (optimizer) estimates costs of each possible execution plans for a SQL statement then the execution plan with the lowest cost is executed. The planner does its best to select the best execution plan, but is not always perfect, since it doesn't take into account some of the data properties or correlations between columns. # Uninstallation

`make uninstall` in the top directory of source tree will uninstall the installed files if you installed from the source tree and it is left available. Setting the environment variable `PATH` may be necessary.

```
$ cd pg_hint_plan-1.x.x
$ su
$ make uninstall
```