



Joining 1 million tables

Hans-Jürgen Schönig

www.postgresql-support.de



Joining 1 million tables

Dedication . . .



- ▶ This talk is dedicated to all broken database designs :)
- ▶ Thank you guys for breaking so much stuff
- ▶ Thank you for inspiring this talk

The goal



- ▶ Creating and joining 1 million tables
- ▶ Pushing PostgreSQL to its limits (and definitely beyond)
- ▶ How far can we push this without patching the core?
- ▶ Being the first one to try this ;)

Creating 1 million tables is easy, no?



- ▶ Here is a script:

```
#!/usr/bin/perl

print "BEGIN;\n";

for (my $i = 0; $i < 1000000; $i++)
{
    print "CREATE TABLE t_tab_$$i (id int4);\n";
}

print "COMMIT;\n";
```



Creating 1 million tables



Who expects this to work?

Well, it does not ;)



```
CREATE TABLE
```

```
CREATE TABLE
```

```
WARNING:  out of shared memory
```

```
ERROR:   out of shared memory
```

```
HINT: You might need to increase  
      max_locks_per_transaction.
```

```
ERROR:   current transaction is aborted, commands ignored  
        until end of transaction block
```

Why does it fail?



- ▶ There is a variable called `max_locks_per_transaction`
- ▶ `max_locks_per_transaction * maximum connections = maximum number of locks`
- ▶ 100 connection \times 64 is by far not enough

A quick workaround



- ▶ Use single transactions
- ▶ Avoid nasty disk flushes on the way

```
SET synchronous_commit TO off;
```

```
time ./create_tables.pl | psql test > /dev/null
real    14m45.320s
user    0m45.778s
sys     0m18.877s
```

Reporting success ...



```
test=# SELECT count(tablename)
        FROM    pg_tables
        WHERE   schemaname = 'public';
 count
-----
1000000
(1 row)
```

- ▶ One million tables have been created

Creating an SQL statement



- ▶ Writing this by hand is not feasible
- ▶ Desired output:

```
SELECT 1
FROM t_tab_1, t_tab_2, t_tab_3, t_tab_4, t_tab_5
WHERE  t_tab_1.id = t_tab_2.id AND
       t_tab_2.id = t_tab_3.id AND
       t_tab_3.id = t_tab_4.id AND
       t_tab_4.id = t_tab_5.id AND
       1 = 1;
```

A first attempt to write a script



```
#!/usr/bin/perl
my $joins = 5;   my $sel = "SELECT 1 ";
my $from = "";  my $where = "";
for (my $i = 1; $i < $joins; $i++)
{
    $from .= "t_tab_$i, \n";
    $where .= " t_tab_" . ($i) . ".id = t_tab_" .
        ($i+1) . ".id AND \n";
}
$from .= " t_tab_$joins ";
$where .= " 1 = 1; ";
print "$sel \n FROM $from \n WHERE $where\n";
```



1 million tables: What it means



- ▶ If you do this for 1 million tables ...

```
perl create.pl | wc  
2000001 5000003 54666686
```

- ▶ 54 MB of SQL is quite a bit of code
- ▶ First observation: The parser works ;)

- ▶ runtimes with PostgreSQL default settings:
 - ▶ $n = 10$: 158 ms
 - ▶ $n = 100$: 948 ms
 - ▶ $n = 200$: 3970 ms
 - ▶ $n = 400$: 18356 ms
 - ▶ $n = 800$: 87095 ms
 - ▶ $n = 1000000$: prediction = 1575 days :)
- ▶ ouch, this does not look linear
- ▶ 1575 days was too close to the conference

GEQO:



- ▶ At some point GEQO will kick in
- ▶ Maybe it is the problem?

```
SET geqo TO off;
```

- ▶ $n = 10$: 158 ms
- ▶ $n = 100$: ERROR: canceling statement due to user request
Time: 680847.127 ms
- ▶ It clearly is not ...

It seems GEQO is needed



- ▶ Let us see how far we can get by adjusting GEQO settings

```
SET geqo_effort TO 1;
```

- n = 10: 158 ms
- n = 100: 916 ms
- n = 200: 1464 ms
- n = 400: 4694 ms
- n = 800: 18896 ms
- n = 10000000: maybe 1 year? :)

=> the conference deadline is coming closer

=> still 999.000 tables to go ...



Trying with 10.000 tables



```
./make_join.pl 10000 | psql test  
ERROR:  stack depth limit exceeded  
HINT:   Increase the configuration parameter  
        max_stack_depth (currently 2048kB), after ensuring  
        the platform's stack depth limit is adequate.
```

- ▶ this is easy to fix

Removing limitations



- ▶ Problem can be fixed easily
OS: `ulimit -s unlimited`
postgresql.conf: set `max_stack_depth` to a very high value

Next step: Exponential planning time



- ▶ Obviously planning time goes up exponentially
- ▶ There is no way to do this without patching
- ▶ As long as we are ways slower than linear it is impossible to join 1 million tables.

This is how PostgreSQL joins tables:



Tables a, b, c, d form the following joins of 2:

[a, b], [a, c], [a, d], [b, c], [b, d], [c, d]

By adding a single table to each group, the following joins of 3 are constructed:

[[a, b], c], [[a, b], d], [[a, c], b], [[a, c], d], [[a, d], b], [[a, d], c], [[b, c], a], [[b, c], d], [[b, d], a], [[b, d], c], [[c, d], a], [[c, d], b]

Likewise, add a single table to each to get the final joins:

[[[a, b], c], d], [[[a, b], d], c], [[[a, c], b], d], [[[a, c], d], b], [[[a, d], b], c], [[[a, d], c], b], [[[b, c], a], d], [[[b, c], d], a], [[[b, d], a], c],
[[[b, d], c], a], [[[c, d], a], b], [[[c, d], b], a]

- ▶ Got it? ;)

A little patch ensures that the list of tables is grouped into “sub-lists”, where the length of each sub-list is equal to `from_collapse_limit`. If `from_collapse_limit` is 2, the list of 4 becomes . . .

[a, b], [c, d]

[a, b] and [c, d] are joined separately, so we only get one “path”:

[[a, b], [c, d]]

The downside:



- ▶ This ensures minimal planning time but is also no real optimization.
- ▶ There is no potential for GEQO to improve things
- ▶ Speed is not an issue in our case anyway. It just has to work “somehow”.

Are we on a path to success?



- ▶ Of course not ;)

The next limitation



- ▶ the join fails again

ERROR: too many range table entries

This time life is easy

Solution: in include/nodes/primnodes.h

```
#define    INNER_VAR                65000
        /* reference to inner subplan */
#define    OUTER_VAR                65001
        /* reference to outer subplan *
#define    INDEX_VAR                65002
        /* reference to index column */
```

to

```
#define    INNER_VAR                2000000
#define    OUTER_VAR                2000001
#define    INDEX_VAR                2000002
```

Next step: RAM starts to be an issue



Tables (k)	Planning time (s)	Memory
2	1	38.5 MB
4	3	75.6 MB
8	31	192 MB
16	186	550 MB
32	1067	1.7 GB

(this is for planning only)

Expected amount of memory: not available ;)

Is there a way out?



- ▶ Clearly: The current path leads to nowhere
- ▶ Some other approach is needed
- ▶ Divide and conquer?

Changing the query



- ▶ Currently there is ...

```
SELECT 1 ...  
FROM tab1, ..., tabn  
WHERE ...
```

- ▶ The problem must be split

CTEs might come to the rescue



- ▶ How about creating WITH-clauses joining smaller sets of tables?
- ▶ At the end those WITH statements could be joined together easily.
- ▶ 1.000 CTEs with 1.000 tables each gives 1.000.000 tables
- ▶ Scary stuff: Runtime + memory Unforeseen stuff

A skeleton query



```
SET enable_indexscan TO off;
SET enable_hashjoin TO off;
SET enable_mergejoin TO off;
SET from_collapse_limit TO 2;
SET max_stack_depth TO '1GB';

WITH cte_0(id_1, id_2) AS (
    SELECT t_tab_0.id, t_tab_16383.id
    FROM t_tab_0, ...
    cte_1 ...
SELECT ... FROM cte_0, cte_1 WHERE ...
```

Finally: A path to enlightenment



- ▶ With all the RAM we had ...
- ▶ With all the patience we had ...
- ▶ With a fair amount of luck ...
- ▶ And with a lot of confidence ...

Finally: It worked ...



- ▶ Runtime: 10h 36min 53 sec
- ▶ The result:

```
?column?
```

```
-----  
(0 rows)
```

Runtime data:



- ▶ The “raw parsing” (by YACC) takes merely 2 seconds. RAM consumption: 933 MB.
- ▶ The “analysis” (this can actually be considered to be the 2nd phase of parsing) took 4144 seconds and 1.4 GB RAM.
- ▶ Planning itself took only 1027 seconds and consumed nearly 13 GB RAM.
- ▶ Execution is an issue

Finally

Thank you for your attention



Cybertec Schönig & Schönig GmbH
Hans-Jürgen Schönig
Gröhrmühlgasse 26
A-2700 Wiener Neustadt

www.postgresql-support.de

