Poor Man's Parallel Processing

PGConf US 2015

Corey Huinker

What is this talk about?

Parallel Processing in Postgres.

What this talk is *really* about?

How wonderfully hackable PostgreSQL is.

Problem: Lack of parallel query in Postgres is hampering adoption.

So, do something about it.

Aren't there available commercial offerings?

Yes, but that's no fun.

What about async sharding (PL/Proxy, etc)?

- Have to build your database around the sharding mechanism.
- Nontechnical people laugh when you say "sharding".

Common technique: Unix Parallel

- Break up your query into smaller queries.
    - One worker handles A-C, next handles D-F...
- Run them separately, combine the results yourself.
    - Ick.

The Goal:

- Something that lets you make something *close* to an ad-hoc query.
- Leveraging multiple CPUs on this machine.
- And maybe that other machine too.
- And have the results coalesced into something that can itself be queried (like a table function).
- Without leaving the query.

Challenges for general parallelism:

- How should I best break up this big query into smaller ones?
  - With no other information, most systems just do a hash distribution.
- At what point would I overload this machine with worker processes?
- Am I just creating a lot of process/network traffic for myself?
  - Poor distribution means lots of interprocess chatter.

PMPP answers *none* of these.

- So why aren't they in PostgreSQL already?
    - Market is littered with problematic parallel half-measures.
    - PostgreSQL Hackers want to get it right the first time.
    - Perfect is the enemy of good in this case.
    - Perfect will be nice when we get it (9.5? 9.6?).
    - In the mean time, here's a half-measure that works in limited circumstances if you're careful.

What does PMPP look like?

When all of your data is on the same machine, but you want to use multiple CPUs:

```
function pmpp.distribute( p_row_type anyelement,
                          p_connection text,
                          p_sql_list text[],
                          p_cpu_multiplier float default 1.0 )
                          returns setof anyelement
```

And for when you want to query multiple machines:

```
function pmpp.distribute( p_row_type anyelement,
                          p_query_manifest in json )
                          returns setof anyelement
```

What does PMPP look like? Zoom in.

When all of your data is on the same machine, but you want to use multiple CPUs:

```
function pmpp.distribute( p_row_type  anyelement,
                          p_connection text,
                          p_sql_list text[],
                          p_cpu_multiplier float default 1.0 )
                          returns setof anyelement
```

polymorphic type-spec

Any postgres DSN string

A list of SQL statements to be executed.

What % of CPUs to allocate.

And for when you want to query multiple machines:

```
function pmpp.distribute( p_row_type anyelement,
                          p_query_manifest in json )
                          returns setof anyelement
```

There's a lot going on here.

Results will match structure of p_row_type

What's this `null::thingamabob` business?

- It's a polymorphic function.
- It gives the shape of the result set that the outer query can expect to receive.
- Is null by convention

Example: single machine queries

```
CREATE TYPE temp_int_row_t (x int);
SELECT
    sum(t.x) as pointless_aggregation
FROM
    pmpp.distribute(null::temp_int_row_t,
                    'dbname=mycurrentdb',
                    ARRAY['select 1',
                          'select 2',
                          'select 3']) t;
```

Result types must be created ahead of time, but all existing table structures are themselves a type.

"row spec"...we'll get to that.

"Loopback" connections are the most common usage. Beware that the main connection may not share permissions with the called one.

Array of query strings.

Example: Query List via Meta-SQL

```
CREATE TYPE temp_int_row_t (x int);
SELECT
    sum(t.x) as overall_rowcount
FROM
    pmpp.distribute(null::temp_int_row_t,
                    'dbname=mycurrentdb',
                    ARRAY(    SELECT
                                  'select count(*) from ' || l.table_name
                              FROM
                                  partition_list l )) t;
```

Just here for an example, you don't have to redefine it every time.

Using SQL to generate SQL is a very powerful way to generate worker commands. The `array()` cast helps visually separate the inner and outer queries.

Example multi-machine query

```
SELECT
    sum(t.x) as overall_rowcount
FROM
    pmpp.distribute(null::temp_int_row_t,
        '[{"connection":"local_dsn", "queries":["SELECT sum(page_loads)
    FROM video_ads WHERE client = ''CUSTOMER1'' AND ad_date >= ''2014-01-
    01''"], "multiplier":"0.5"},{"connection":"archive_dsn", "queries":
    ["SELECT sum(page_loads) FROM video_ads WHERE client = ''CUSTOMER1''
    AND ad_date < ''2014-01-01''"],"workers":"2"}]'::jsonb) t;
```

WHA???

Wait, what was that JSON about?

```
[{"connection":"local_dsn",
  "queries":[
     "SELECT sum(page_loads) FROM video_ads
     WHERE client = 'CUSTOMER1'
     AND ad_date >= '2014-01-01'"],
  "multiplier":"0.5"},
{ "connection":"archive_dsn",
  "queries":[
    "SELECT sum(page_loads) FROM video_ads
    WHERE client = 'CUSTOMER1'
    AND ad_date < '2014-01-01'"],
  "workers":"2"}]
```

Each section has connection info, like the local version.

We'd normally expect a lot of queries in at least one of the sections, but this is just an example.

We know it has PMPP installed and we want to use AT MOST half the CPUs.

The queries have to all have the same shape of result set.

Might not have PMPP installed, might not even be real PostgreSQL...

Did you try anything other than polymorphic functions? - Yes: JSON

```
SELECT
    sum((t.json_data->>'row_count')::bigint) as row_count
FROM
    mpp_dist_json(
        ARRAY(SELECT
                'select count(*) as row_count from partitions.'
                  || partition_name
              FROM
                partition_metadata_table
              WHERE
                table_name = 'my_partitioned_table')
        ) t;
```

Re-composition acrobatics and typecasting

Project name has changed over time

Meta-SQL is basically the same.

It's not the prettiest, and the decompose-recompose overhead increases with the number of columns.

Did you try anything other than polymorphic functions? - HSTORE

```
SELECT
    sum((t.hstore_data->'row_count')::bigint)
FROM
    pmpp_dist_hstore(
        array(SELECT
                'select count(*) as row_count from partitions.'
                  || partition_name
              FROM
                partition_metadata_table
              WHERE
                table_name = 'my_partitioned_table')
                ) t;
```

Basically the same tradeoffs as JSON/JSONB.

What's under the hood?

- DBLINK extension
  - `dblink_send_query()` and `dblink_get_result()` async functions
  - This module lacked ability to do polymorphic result sets.
    - So I wrote a patch for that.
    - Ain't hackability great?
- A pg_attribute query to create table spec
  - `FROM dblink_get_result(x)` **`AS t(col1 int, …)`**
  - Query has to be constructed dynamically once, and re-run once per subquery.
  - PL/PGSQL lacks a `PREPARE` statement
    - Thought about moving to plv8 or C.
  - Will still need this until DBLINK supports polymorphism.

Under the hood: pg_attribute query

```
WITH x as (
    select  a.attname || ' ' || pg_catalog.format_type(a.atttypid,
             a.atttypmod) as sql_text
    from    pg_catalog.pg_attribute a
    where   a.attrelid = pg_typeof(p_row_type)::text::regclass
    and     a.attisdropped is false
    and     a.attnum > 0
    order by a.attnum )
SELECT  format('select * from dblink_get_result($1) as t(%s)',
                 string_agg(x.sql_text,','))
INTO    fetch_results_query
FROM    x;
```

Runtime: about 1ms.

What's under the hood?

- PL/PGSQL
  - one `FOR LOOP`
    - really just there to look for failures in initial query distribution.
  - and one `WHILE LOOP`
    - looking for queries that have finished, launching new queries as old ones complete, closing down connections
      - `pg_sleep()` with exponential backoff
  - A surprising amount of iteration can be handled in SQL itself.
- temp tables for work queue management, connection management.
  - Wasn't appreciably slower than PL/PGSQL arrays and state variables.
  - Cleaner code, likely very easy to port to C/v8, etc.

How do you know how many workers to spawn?

By cheating! Hijack the `copy` command to invoke a command line.

```
create temporary table nproc_result (nproc integer);
copy nproc_result from program ' nproc';
select
    format('$$ select greatest(1,(p_multiplier * %s)::integer)$$',
            nproc) as nproc_sql
from
    nproc_result
\gset

create or replace function pmpp.num_cpus(p_multiplier in float default
1.0) returns integer
language sql immutable as  :nproc_sql;
```

Sooooo not portable.

Using SQL to generate SQL again

Saves each column of the one-row result set as a same-named variable

Using PSQL vars in SQL definitions.
No $$ quotations needed.

So now you've got an immutable function: ultra-low overhead.

How are you using it?

- ETL
    - Partition refresh in place of python & `multiprocessing`
    - Index Rebuilds
- Deployment scripts
    - Partition creation
- Big-Question queries
    - our data is timeseries, so asking questions across all time can be compute intensive. Partial sums make it more manageable.
- In Development
    - Three-tiered data storage
        - in-memory cache accessed via custom FDW
        - Vertica for recent data
        - Redshift for archive data

So many questions!

Q. So this would put passwords in the clear, huh?

- Yup, anyone with pg_stat_activity visibility on the *initiating* machine could see them.

Q. How do you know how many connections are available?

- You don't! (See: Running With Scissors)

Q. What if the other machine doesn't have pmpp installed?
What if the other machine isn't a "real" postgres (Vertica, Redshift)?

- Use the num_workers parameter instead of the multiplier.

Q. What's a good multiplier to use?

- 1.0 on AWS EC2s with local SSD drives.
    - Yes, cpu multipliers on Oracle are usually 2x to 4x the number of CPUs.
    - Our queries are very sum-oriented.

Future Direction

1. Put PMPP on PGXN
2. CPU detection extension so that we don't rely on nproc existing anymore.
3. Get patch to DBLINK accepted into 9.5.
4. Become obsolete.