Poor Man's Parallel Processing

PgDay Austin 2016
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.

Didn't they add parallel in 9.6?

- Sorta
- Doesn't support CTEs, INSERT INTO SELECT, CREATE TABLE AS, etc
- It will get better, though.

Aren't there available commercial offerings?

Yes, but that's no fun.

What about async sharding (PL/Proxy, Citus, XL, 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.
    - Progress is slow.
    - More options coming in v10 (async_fdw, etc).
    - 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:

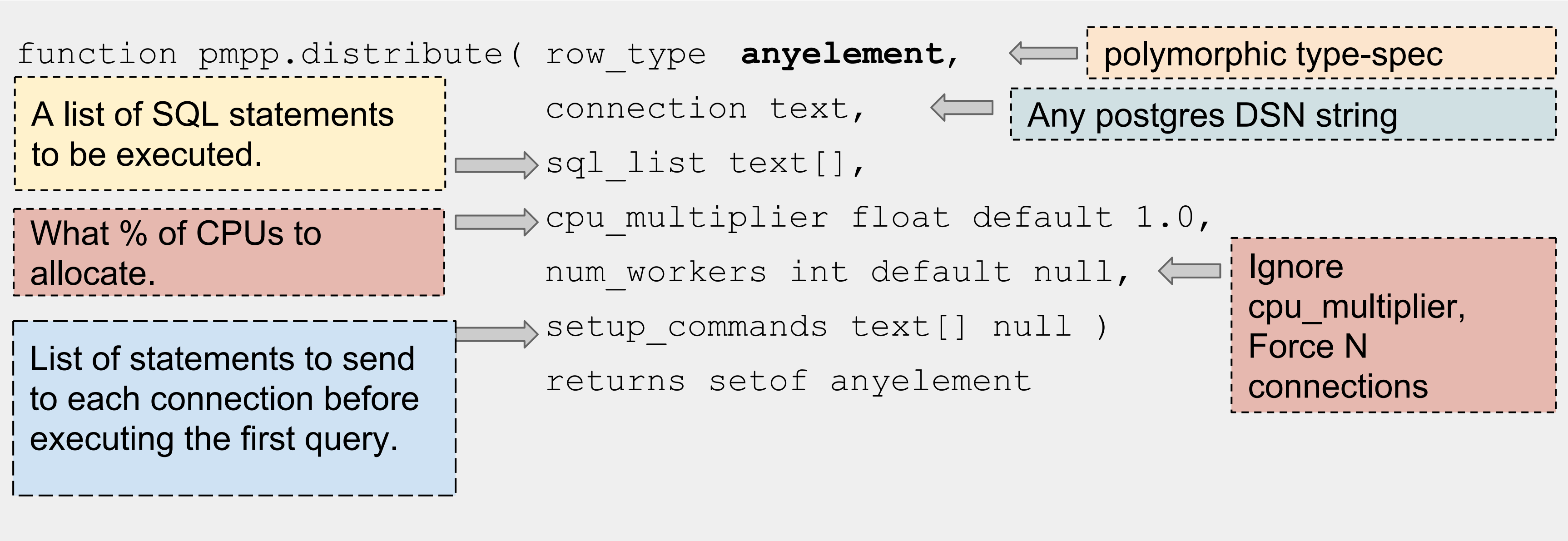
```
function pmpp.distribute( row_type anyelement, connection text,
       sql_list text[], cpu_multiplier float default 1.0,
       num_workers int default null, setup_commands text[] default null)
                          returns setof anyelement
```

And for when you want to query multiple machines:

```
function pmpp.distribute( row_type anyelement,
                          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( row_type  anyelement,
                          connection text,
                          sql_list text[],
                          cpu_multiplier float default 1.0,
                          num_workers int default null,
                          setup_commands text[] null )
                          returns setof anyelement
```

polymorphic type-spec

Any postgres DSN string

A list of SQL statements to be executed.

What % of CPUs to allocate.

Ignore cpu_multiplier, Force N connections

List of statements to send to each connection before executing the first query.

What does PMPP look like? Zoom in.

And for when you want to query multiple machines:

```
function pmpp.distribute( row_type anyelement,
                          query_manifest in json )
                          returns setof anyelement
```

Results will match structure of row_type

There's a lot going on here.

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,
                    'my_fdw_conn_name',
                    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.

Array of query strings.

Using a defined FOREIGN SERVER with a user mapping is one way to hide connection string details.

Example: SETOF RECORD mode

```
SELECT
    sum(t.x) as pointless_aggregation
FROM
    pmpp.distribute('[{"connetion": "my_fdw_name", "queries": ["SELECT 1", "SELECT 2", "SELECT 3"]}]'::jsonb) as t(x integer);
```

- Record format is determined by the record spec
- Wasn't possible to do it this way in pl/pgsql
- Had to implement in C/libpq directly.

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;
```

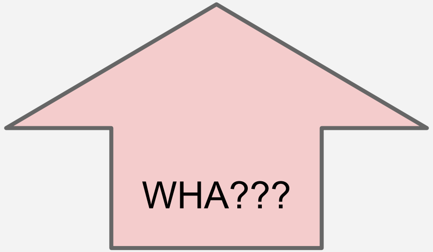
Just here for an example, you don't have to redefine it every time.

Be careful with "loopback" connections. They may not have the same permissions & search_path of your main session.

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?

- libpq and async queries (written in C)
- earlier versions used the 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 (it got rejected).
        - Ain't hackability great?
- the polymorphic type becomes the rsinfo for the set returning function.
- child data sets are passed on to the parent data set, doing type coercion if attr oids don't match
- Can use libpq text or binary mode.
    - Binary is faster.
    - Binary doesn't work with Vertica, Redshift

What's under the hood?

- If remote system has pmpp installed, then local system will ask how many CPUs it has and multiply that by cpu_multiplier.
- Systems that don't have pmpp installed **must** have num_workers specified for their connections.
- Never creates more connections that it has queries to dispatch
- Dispatches a query to each connection after it is connected and has run all setup_commands entries. Only then does it move on to the next connection.
- All worker data structures kept in memory in C. The dblink version used temp tables.
- Can be used on a read replica if the polymorphic return types already exist.
- new

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 do you know how many workers to spawn? - libpq

- Simple libpq function invoking sysconf(_SC_NPROCESSORS_ONLN));
- Lots of things are easier once you get to know libpq

How are you using it?

- ETL
  - Partition refresh in place of python & `multiprocessing`
  - Index Rebuilds
- Initial partition creation in Deployment scripts
- 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
  - Four-tiered data storage
    - in-memory cache accessed via custom FDW
    - PostgreSQL for most recent N days.
    - Vertica for recent data N+ days to a year old.
    - Redshift for data more than a year old

So many questions!

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

- Yup, Which is why it's a good idea to use names of foreign servers with user mappings.

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, and usually have hundreds of sum columns.

Future Direction

1. Possibly autodetect whether binary mode can be used without having to specify it (depends a lot on usage of non-default datatypes in result set).
2. Find best balance of pg_sleep/mu-sleep/check-interrupts for having the master connection wait when all children are busy.
3. More ways to invoke with SETOF RECORD rather than polymorphic invocations.
4. Helper functions for constructing complicated query_manifest structures and/or JSONB.
5. Benchmark against upcoming async mode in foreign data wrappers, which should accomplish the same thing but with less control of the remote query plan.