Reaching 1 billion rows / second

Hans-Jürgen Schönig
www.postgresql-support.de

Reaching a milestone

Traditional PostgreSQL limitations

- Traditionally:
  - We could only use 1 CPU core per query
  - Scaling was possible by running more than one query at a time
  - Usually hard to do

PL/Proxy: The traditional way to do it

- PL/Proxy is a stored procedure language to scale out to shards.
- Worked nicely for OLTP workloads
- Somewhat usable for analytics
  - A LOT of manual work

PL/Proxy: The future

- Still ok for OLTP
- Certainly not the way to scale out in the future
- Too much manual work
- Not transparent
- Not cool enough

On the app level

- Doing scaling on the app level
  - A lot of manual work
  - Not cool enough
  - Needs a lot of development
  - Why use a database if work is still manual?
- Solving things on the app level is certainly not an option

The goal

Our goal on the PostgreSQL level

- Import massive amounts of data
- Run typical aggregates
- Process 1 billion rows in less than a second
- Scale out to as many nodes as needed

Coming up with a data structure

- We tried to keep that simple:

```
node=# \d t_demo
           Table "public.t_demo"
 Column |  Type    | Collation | Nullable |
--------+----------+-----------+----------+
 id     | serial   |           | not null |
 grp    | integer  |           |          |
 data   | real     |           |          |
Indexes:
    "idx_id" btree (id)
```

The query

```
SELECT   grp, count(data)
FROM     t_demo
GROUP BY 1;
```

Single server performance

Tweaking a simple server

- The main questions are:
  - How much can we expect from a single server?
  - How well does it scale with many CPUs?
  - How far can we get?

PostgreSQL parallelism

- Parallel queries have been added in PostgreSQL 9.6
  - It can do a lot
  - It is by far not feature complete yet
- Number of workers will be determined by the PostgreSQL optimizer
  - We do not want that
  - We want ALL cores to be at work

Adjusting CPU core usage

- Usually the number of processes per scan is derived from the size of the table

```
test=# SHOW min_parallel_relation_size ;
 min_parallel_relation_size
----------------------------
 8MB
(1 row)
```

- One process is added if the tablesize triples

Overruling the planner

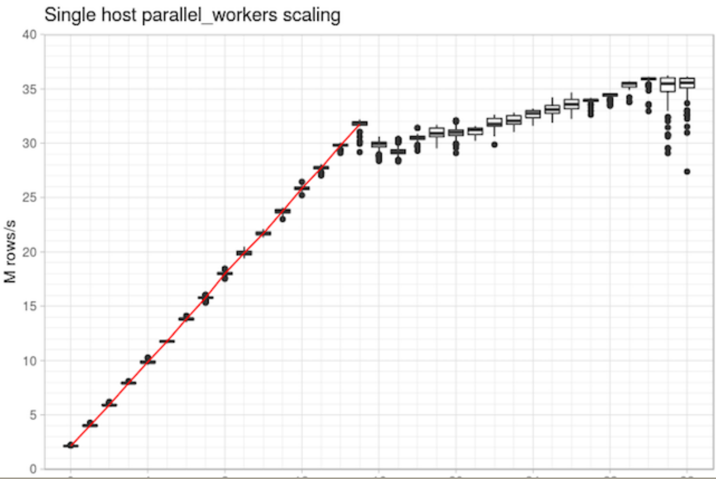
- We could never have enough data to make PostgreSQL go for 16 or 32 cores.
- Even if the value is set to a couple of kilobytes.
- The default mechanism can be overruled:

```
test=# ALTER TABLE t_demo
    SET (parallel_workers = 32);
ALTER TABLE
```

Making full use of cores

- How well does PostgreSQL scale on a single box?
- For the next test we assume that I/O is not an issue
  - If I/O does not keep up, CPU does not make a difference
  - Make sure that data can be read fast enough.
- Observation: 1 SSD might not be enough to feed a modern Intel chip

Single node scalability (1)



Single node scalability (2)

- We used a 16 core box here
- As you can see, the query scales up nicely
- Beyond 16 cores hyperthreading kicks in
  - We managed to gain around 18%

Single node scalability (3)

- On a single Google VM we could reach close to 40 million rows / second
- For many workloads this is already more than enough
- Rows / sec will of course depend on type of query

Moving on to many nodes

The basic system architecture (1)

- We want to shard data to as many nodes as needed
- For the demo: Place 100 million rows on each node
  - We do so to eliminate the I/O bottleneck
  - In case I/O happens we can always compensate using more servers
- Use parallel queries on each shard

Testing with two nodes (1)

```
explain SELECT grp, COUNT(data) FROM t_demo GROUP BY 1;
 Finalize HashAggregate
   Group Key: t_demo.grp
   ->  Append
         ->  Foreign Scan  (partial aggregate)
         ->  Foreign Scan  (partial aggregate)
         ->  Partial HashAggregate
               Group Key: t_demo.grp
               ->  Seq Scan on t_demo
```

Testing with two nodes (2)

- Throughput doubles as long as partial results are small
- Planner pushes down stuff nicely
- Linear increases are necessary to scale to 1 billion rows

Preconditions to make it work (1)

- postgres_fdw uses cursors on the remote side
  - cursor_tuple_fraction has to be set to 1 to improve the planning process
  - set fetch_size to a large value
- That is the easy part

Preconditions to make it work (2)

- We have to make sure that all remote nodes work at the same time
- This requires “parallel append and async fetching”
  - All queries are sent to the many nodes in parallel
  - Data can be fetched in parallel

Preconditions to make it work (3)

- PostgreSQL could not be changed without substantial work being done recently
  - Traditionally joins had to be done BEFORE aggregation
  - This is a showstopper for distributed aggregation because all the data has to be fetched from the remote host before aggregation
- Kyotaro Horiguchi fixed, which made our work possible
  - This was a HARD task !

Preconditions to make it work (4)

- Easy tasks:
  - Aggregates have to be implemented to handle partial results coming from shards
  - Code is simple and available as extension

Parallel execution on shards is now possible

- Dissect aggregation
- Send partial queries to shards in parallel
- Perform parallel execution on shards
- Add up data on main node

Final results

```
node=# SELECT grp, count(data) FROM t_demo GROUP BY 1;
 grp |   count
-----+------------
   0 | 320000000
   1 | 320000000
...
   9 | 320000000
(10 rows)
 Planning time: 0.955 ms
 Execution time: 2910.367 ms
```

Hardware used

- We used 32 boxes (16 cores) on Google
- Data was in memory
- Adding more servers is EASY

Future ideas

- JIT compilation will speed up execution
- More parallelism for more executor nodes
- General speedups (tuple deforming, etc.)
- In the future FEWER cores will be needed to achieve similar results

Contact us

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

www.postgresql-support.de

Follow us on Twitter: @PostgresSupport