intro to benchmarking with `pgbench`

Melanie Plageman (Microsoft)

Load data:

```
pgbench -i --scale
```

Run pgbench:

```
pgbench [options]
```

it is

- Common performance language for hackers
- Convenience tool

Use it to:

- Compare two versions of Postgres
- Compare two Postgres configurations
- Test the performance of your server or instance

it isn’t

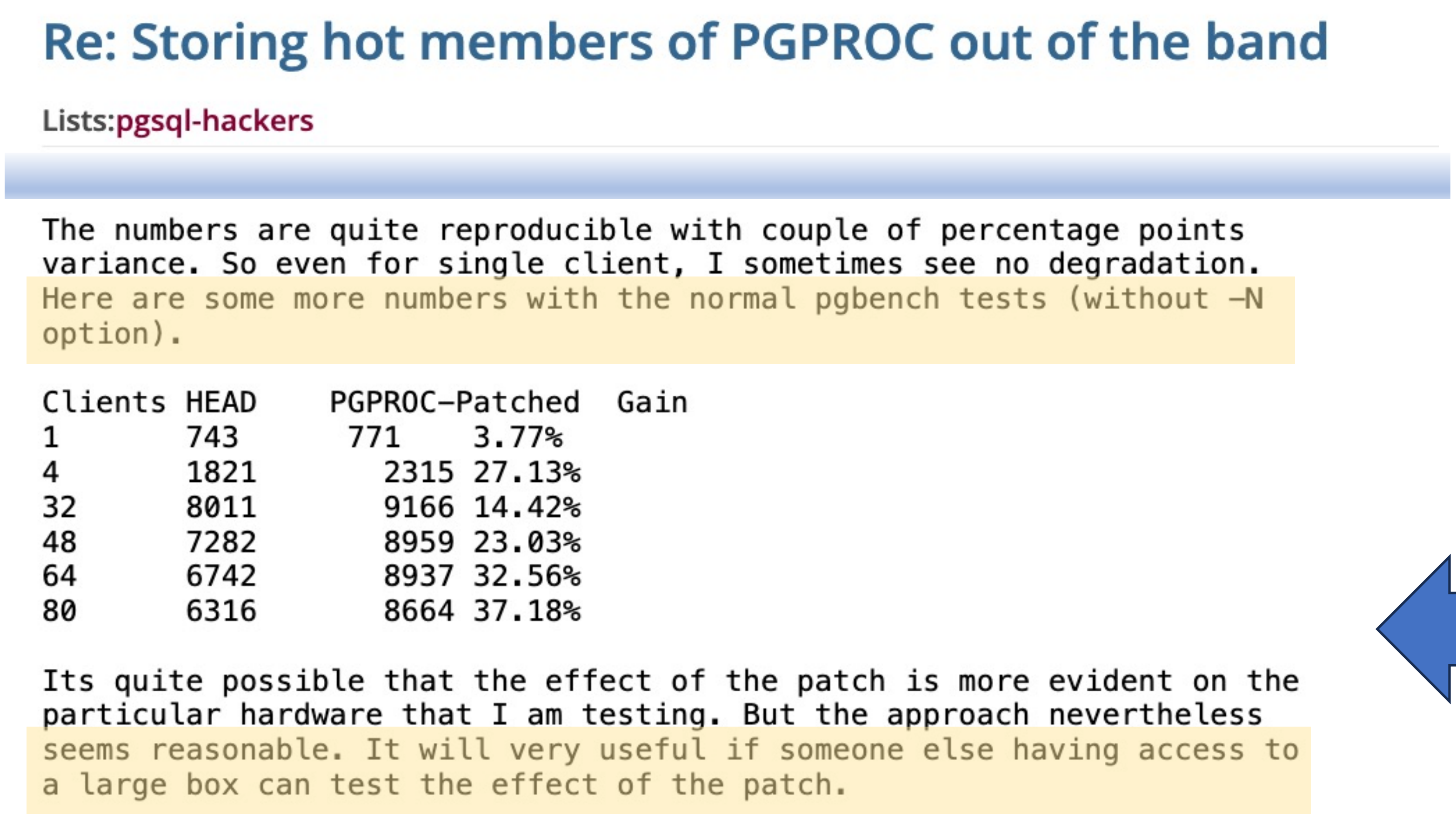
- Real schema or workload performance analysis tool
- Database comparison tool

Don’t use it to:

- Test the performance of your database schema or specific workload
- Test patches without a specific hypothesis

performance language for Postgres hackers

- other developers can run your benchmarks
- common understanding of behavior of built-in scripts and options
- reproducible benchmarks



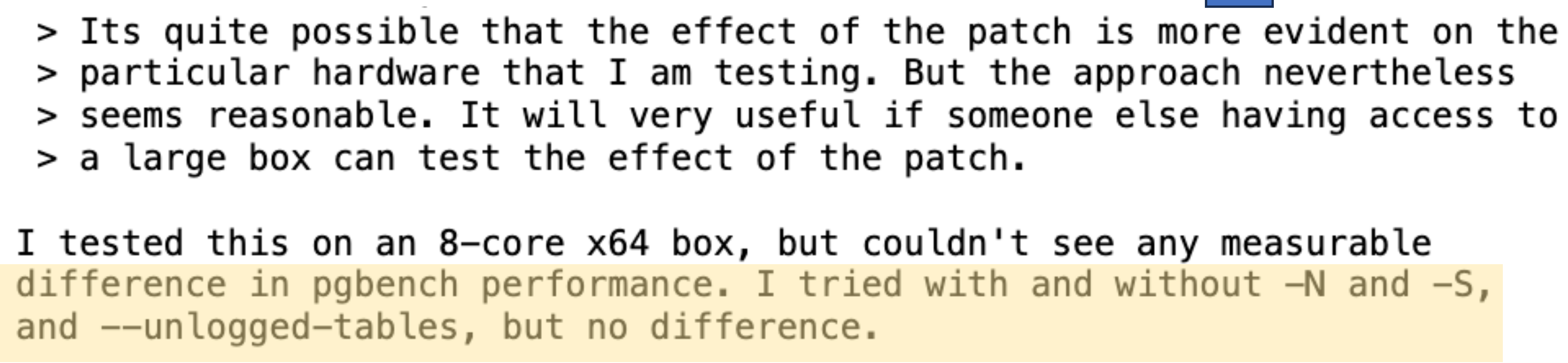
Re: Storing hot members of PGPROC out of the band

Lists:pgsql-hackers

The numbers are quite reproducible with couple of percentage points variance. So even for single client, I sometimes see no degradation. Here are some more numbers with the normal pgbench tests (without -N option).

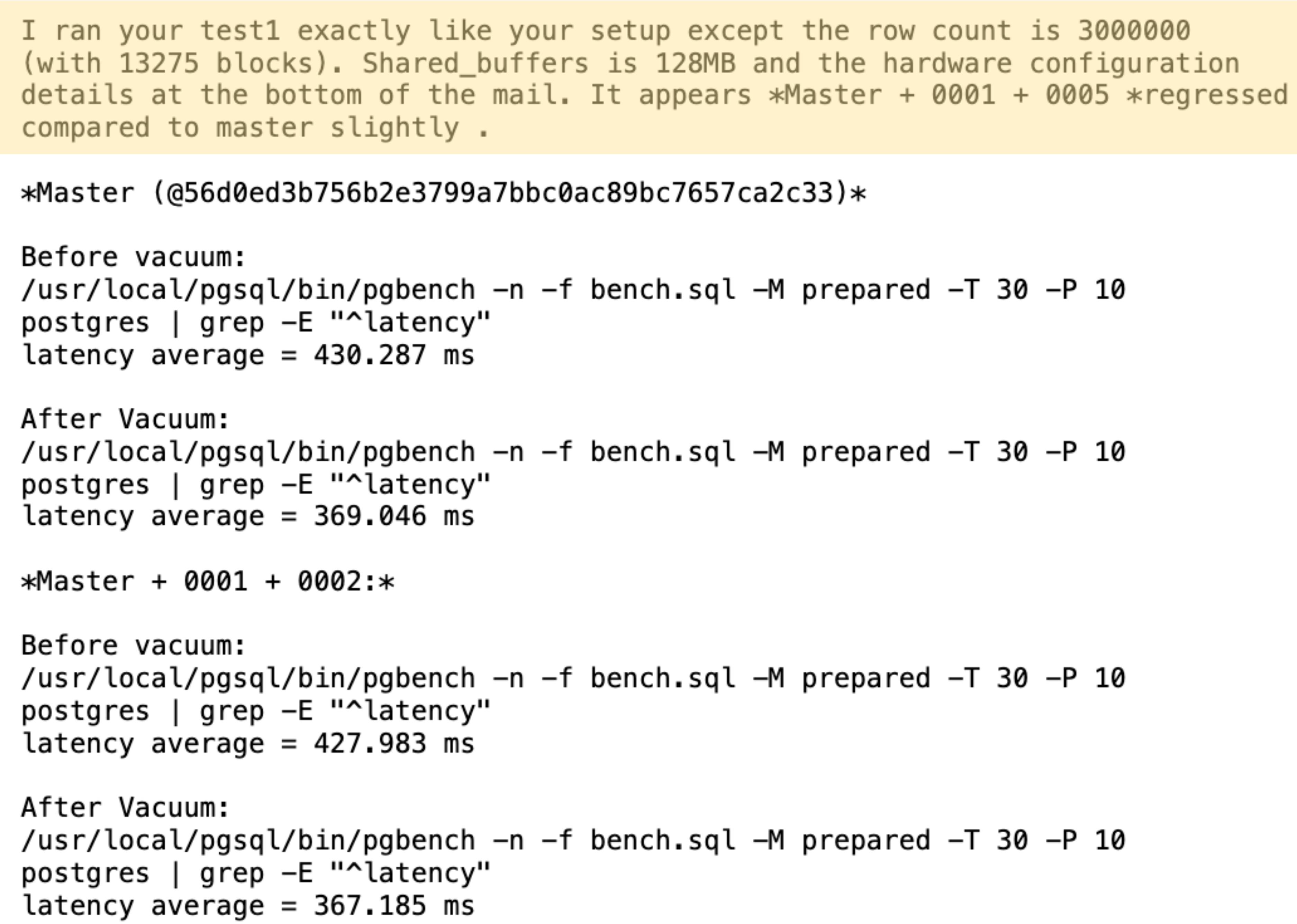
Clients	HEAD	PGPROC-Patched	Gain
1	743	771	3.77%
4	1821	2315	27.13%
32	8011	9166	14.42%
48	7282	8959	23.03%
64	6742	8937	32.56%
80	6316	8664	37.18%

Its quite possible that the effect of the patch is more evident on the particular hardware that I am testing. But the approach nevertheless seems reasonable. It will very useful if someone else having access to a large box can test the effect of the patch.



> Its quite possible that the effect of the patch is more evident on the
> particular hardware that I am testing. But the approach nevertheless
> seems reasonable. It will very useful if someone else having access to
> a large box can test the effect of the patch.

I tested this on an 8-core x64 box, but couldn't see any measurable difference in pgbench performance. I tried with and without -N and -S, and --unlogged-tables, but no difference.



> I ran your test1 exactly like your setup except the row count is 3000000 (with 13275 blocks). Shared_buffers is 128MB and the hardware configuration details at the bottom of the mail. It appears *Master + 0001 + 0005 *regressed compared to master slightly .

```
*Master (@56d0ed3b756b2e3799a7bbc0ac89bc7657ca2c33)*

Before vacuum:
/usr/local/pgsql/bin/pgbench -n -f bench.sql -M prepared -T 30 -P 10
postgres | grep -E "^latency"
latency average = 430.287 ms

After Vacuum:
/usr/local/pgsql/bin/pgbench -n -f bench.sql -M prepared -T 30 -P 10
postgres | grep -E "^latency"
latency average = 369.046 ms

*Master + 0001 + 0002:*

Before vacuum:
/usr/local/pgsql/bin/pgbench -n -f bench.sql -M prepared -T 30 -P 10
postgres | grep -E "^latency"
latency average = 427.983 ms

After Vacuum:
/usr/local/pgsql/bin/pgbench -n -f bench.sql -M prepared -T 30 -P 10
postgres | grep -E "^latency"
latency average = 367.185 ms
```

convenience tool for developers

- Replicate specific performance scenarios with load and run options
- Generate and collect output stats

generate standard table data with options

- `--foreign-keys`
- `--index-tablespace=index_tablespace`
- `--partition-method=NAME`
- `--partitions=NUM`
- `--tablespace=tablespace`
- `--unlogged-tables`
- `--fillfactor=fillfactor`

control `pgbench` run

- `--latency-limit=limit`
- `--protocol=querymode`
- `--max-tries=number_of_tries`
- `--rate=rate`
- `--client=clients`

detailed progress output and stats

- `--log`
- `--progress=sec`
- `--report-per-command`
- `--aggregate-interval=seconds`
- `--sampling-rate=rate`

`pgbench` options and workloads

important `pgbench` options

- Load options
  - `--scale`
- Run options
  - `--protocol`
  - `--client`
  - `--builtin`
  - `--file`
  - `--time/--transactions`

--scale

how much data is loaded into default tables

- branches = 1x
- tellers = 10x
- accounts = 100,000x

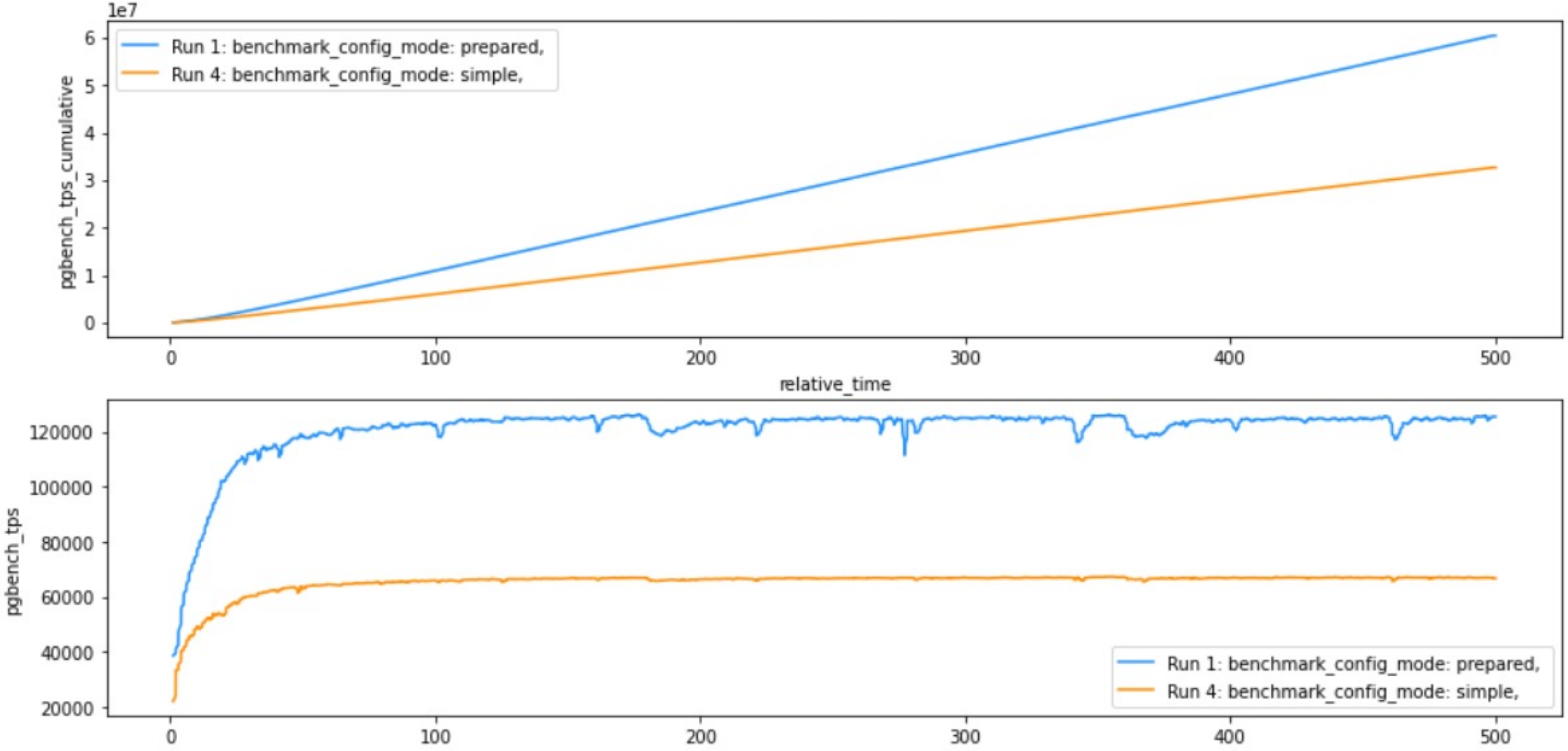
`--protocol` (query mode)

simple: query text sent to the server and parsed on every execution

extended: query execution split into parse, bind, and execute but doesn’t save and reuse prepared statements

prepared: saves and reuses the prepared statements from the first execution

Prepared query mode performs better



--client

- Number of concurrent Postgres sessions

Specify run duration

`--time`

`--transactions`

--builtin and --file

--builtin

- tpcb-like
- select-only
- simple-update

custom script with --file

- testing specific scenarios
- testing on specific data

what do the built-in scripts actually do?

TPCB-like

```
1.BEGIN;

2.UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;

3.SELECT abalance FROM pgbench_accounts WHERE aid = :aid;

4.UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;

5.UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;

6.INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
       VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);

1.END;
```

Simple UPDATE

1.BEGIN;

2.UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;

3.SELECT abalance FROM pgbench_accounts WHERE aid = :aid;

~~4.UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;~~

~~5.UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;~~

6.INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
      VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);

1.END;

SELECT-only

~~1.BEGIN;~~

~~2.UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;~~

3.SELECT abalance FROM pgbench_accounts WHERE aid = :aid;

~~4.UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;~~

~~5.UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;~~

~~6.INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)~~
~~VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);~~

~~1.END;~~

pgbench variables

```
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

Actual SELECT-only workload commands

```
\set aid random(1, naccounts * :scale)
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

naccounts is hard-coded to 100,000

`pgbench` meta commands

- **`\set`** ***`varname expression`***
- `\gset [`*`prefix`*`] \aset [`*`prefix`*`]`
- `\if` *`expression`*
  `\elif` *`expression`*
  `\else`
  `\endif`
- `\sleep` *`number`* `[ us | ms | s ]`
- `\setshell` *`varname command`* `\`
  `[` *`argument`* `... ]`
- `\shell` *`command`* `[` *`argument`* `... ]`
- `\start|endpipeline`

- similar syntax to equivalent psql commands
- does not use psql-style SQL interpolation for variables

`\set` *`varname expression`*

- Sets variable *varname* to a value calculated from *expression*
- may contain
  - NULL
  - Boolean, integer, or double constants
  - variable references
  - pgbench built-in operators
  - pgbench built-in function calls
  - SQL CASE generic conditional expressions and parentheses

`\gset [`*`prefix`*`] \aset [`*`prefix`*`]`

- \gset
  - stores columns of preceding SQL query (returning one row) into variables named after column names preceded with prefix
  - valid in psql
- \aset
  - stores columns in all preceding combined SQL queries (separated by \;) into variables named after column names preceded with prefix
  - not valid in psql

\gset Use case

```
table1 (id serial, time default now(), status int)

INSERT INTO table1 (status)
    VALUES (1) RETURNING id AS target \gset

UPDATE table1 SET status = 2 WHERE id = :target;
```

```
\set aid random(1, naccounts * :scale)
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

built-in functions for pgbench meta commands

- `abs(), exp(), ln(), mod(), pow(), pi(), sqrt(), greatest(), least()`
- `double(), int()`
- `hash(), hash_fnv1a(), hash_murmur2()`
- `permute(),` **`random()`**`, random_exponential(), random_gaussian(), random_zipfian()`
- `debug()`

built-in TPCB-like access distribution

```
\set aid random(1, naccounts * :scale)
\set bid random(1, nbranches * :scale)
\set tid random(1, ntellers * :scale)
\set delta random(-5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
        VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

TPCB-like variant with Gaussian access distribution

```
\set aid random_gaussian(1, naccounts * :scale, 6)
\set bid random_gaussian(1, nbranches * :scale, 6)
\set tid random_gaussian(1, ntellers * :scale, 6)
\set delta random_gaussian(-5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
        VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

```
\set aid random(1, naccounts * :scale)
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

automatic variables

- `client_id`
- `default_seed`
- `random_seed`
- **`scale`**

passing and overriding variables

- -D

automatic variables

- **`client_id`**
- `default_seed`
- `random_seed`
- `scale`

`client_id` use case

```
for each client
    psql -c "CREATE TABLE table_$client_id(…)"

COPY table_:client_id FROM 'copysource';
```

pgbench output

pgbench output

- load summary
- run summary
- --progress
- --log

summary output

Load Summary

```
done in 0.86 s
drop tables 0.07 s,
create tables 0.11 s,
client-side generate 0.25 s,
vacuum 0.21 s,
primary keys 0.23 s
```

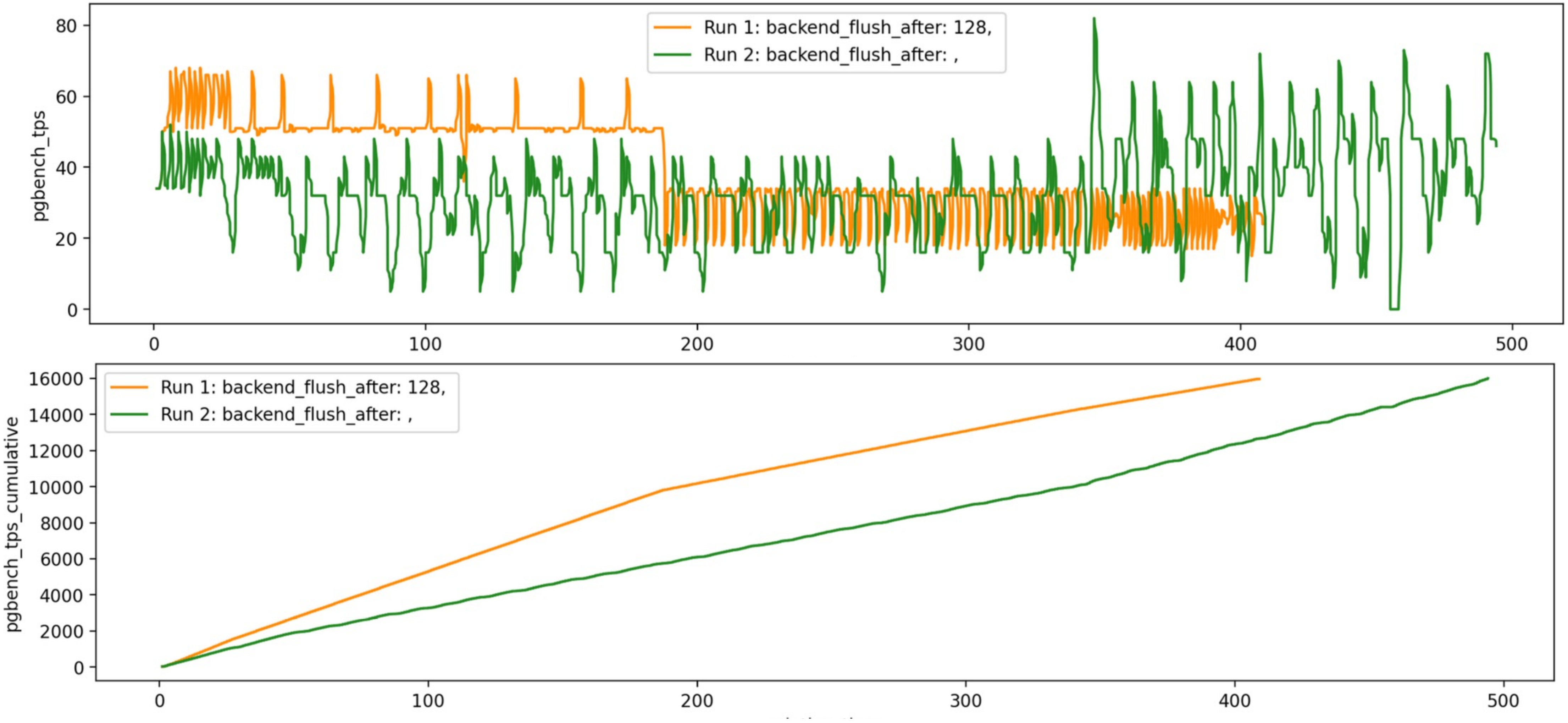
Run Summary

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
duration: 4 s
number of transactions actually processed: 158
latency average = 25.341 ms
latency stddev = 3.607 ms
initial connection time = 4.897 ms
tps = 39.459091 (without initial connection time)
```

--progess [interval]

```
progress: 1.0 s, 37.9 tps, lat 25.753 ms stddev 4.787
progress: 2.0 s, 41.0 tps, lat 24.850 ms stddev 1.014
progress: 3.0 s, 40.1 tps, lat 24.569 ms stddev 0.900
progress: 4.0 s, 37.9 tps, lat 26.263 ms stddev 5.225
```

viewing performance metrics over time



designing a benchmark to test the performance of a feature

picking a workload

- patch to change the default bulk write ring buffer size
- testing this with pgbench built-in TPCB-like does not exercise the code

```
BufferAccessStrategy
GetAccessStrategy(…)
...
switch (btype)
{
    case BAS_NORMAL:
            return NULL;
    case BAS_BULKREAD:
            ring_size_kb = 256;
            break;
    case BAS_BULKWRITE:
            ring_size_kb = 16 * 1024;
            break;
    case BAS_VACUUM:
            ring_size_kb = 256;
            break;
}
```

evaluating the results

- summary output
- `--progress`
- `--log`

```
pgbench -c 1 -M prepared -P 1 -t 50\
    -f <(echo “COPY table(…) FROM 'copysource’”)
```

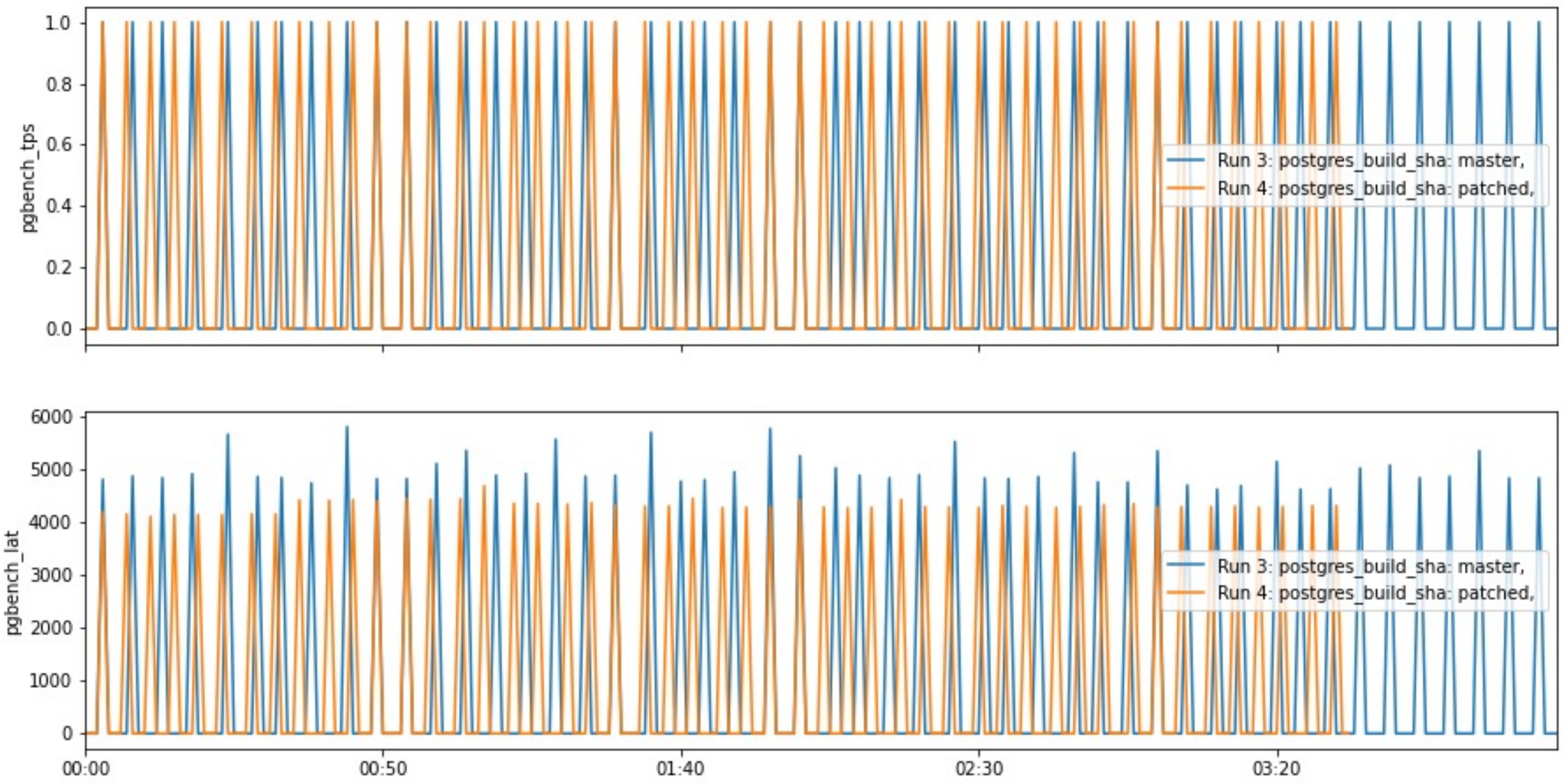
output summary patched

```
pgbench (17devel)
scaling factor: 1
query mode: prepared
number of clients: 1
number of threads: 1
maximum number of tries: 1
number of transactions per client: 50
number of transactions processed: 50/50
number of failed transactions: 0
latency average = 4316.597 ms
latency stddev = 101.553 ms
initial connection time = 1.653 ms
tps = 0.231663
```

output summary unpatched

```
pgbench (17devel)
scaling factor: 1
query mode: prepared
number of clients: 1
number of threads: 1
maximum number of tries: 1
number of transactions per client: 50
number of transactions processed: 50/50
number of failed transactions: 0
latency average = 5015.483 ms
latency stddev = 314.739 ms
initial connection time = 1.124 ms
tps = 0.199382
```

COPY –progress often isn’t useful



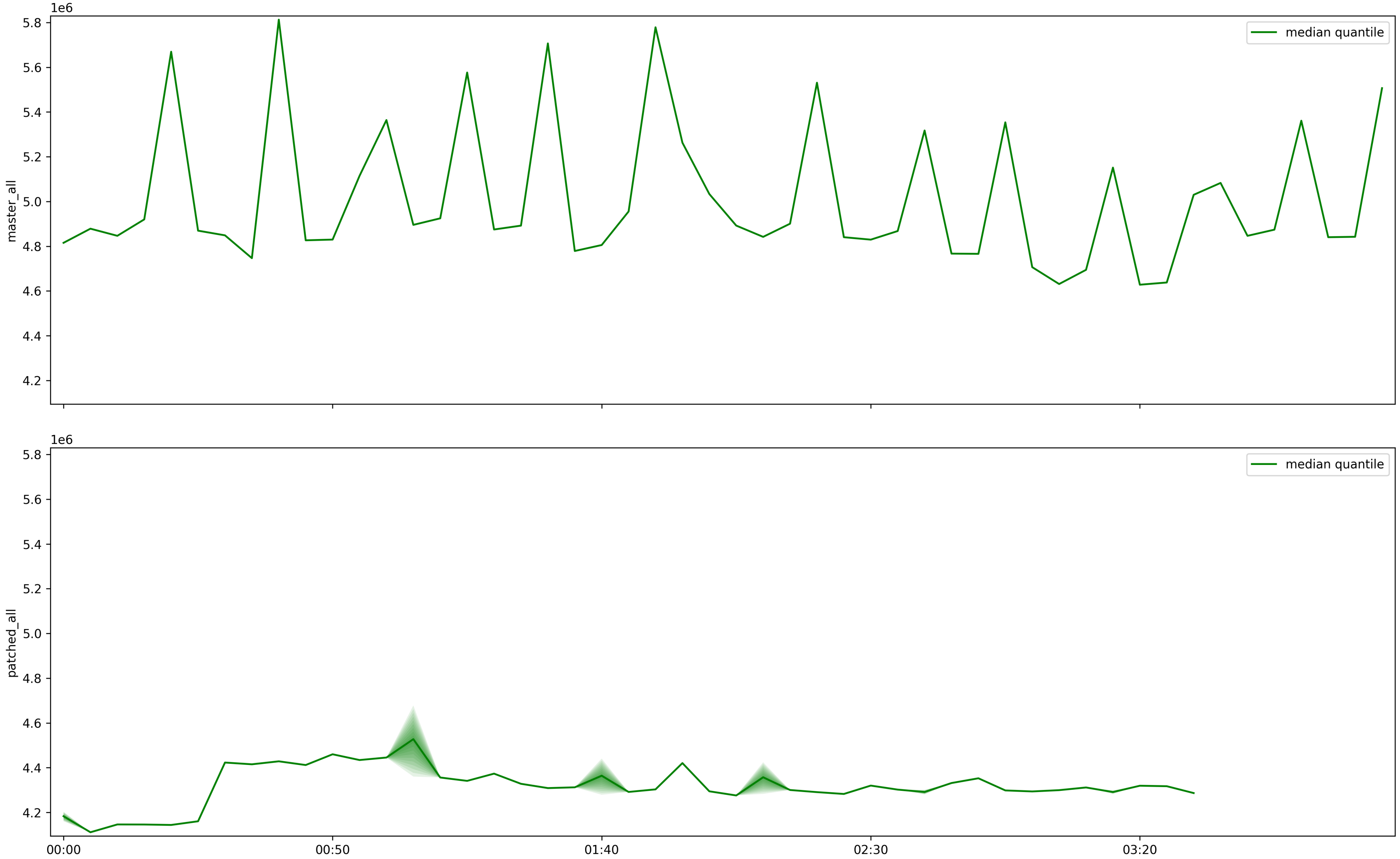
--log

```
client_id|transaction_no|   time  |script_no| time_epoch |  time_us
        0|             1|  4815382|        0|  1696097811|    347611
        0|             2|  4878592|        0|  1696097816|    226217
        0|             3|  4846731|        0|  1696097821|     72959
        0|             4|  4920091|        0|  1696097825|    993061
        0|             5|  5669546|        0|  1696097831|    662617
        0|             6|  4869671|        0|  1696097836|    532298
        0|             7|  4848832|        0|  1696097841|    381138
        0|             8|  4746861|        0|  1696097846|    128010
        0|             9|  5812907|        0|  1696097851|    940926
        0|            10|  4826624|        0|  1696097856|    767560
        0|            11|  4829888|        0|  1696097861|    597458
        0|            12|  5114546|        0|  1696097866|    712011
        0|            13|  5364114|        0|  1696097872|     76135
```

COPY benchmark results from `--log`

	patched	master
total time (ms)	215,829	250,774
average time (ms)	4,316	5,015
median time (ms)	4,302	4,876
minimum time (ms)	4,111	4,628
maximum time (ms)	4,695	5,812
standard dev time (ms)	101	314

Plotting `COPY --log`



Postgres and OS Configuration

factors affecting benchmark results

- OS configuration
  - OS page size
  - block device settings
- Hardware considerations
  - age of storage device
  - CPU power management settings
- Filesystem choice and mount options
- Postgres compile options
- Postgres configuration
- Run steps
  - CPU affinity
  - initdb before benchmark

configure the options that matter to your benchmark

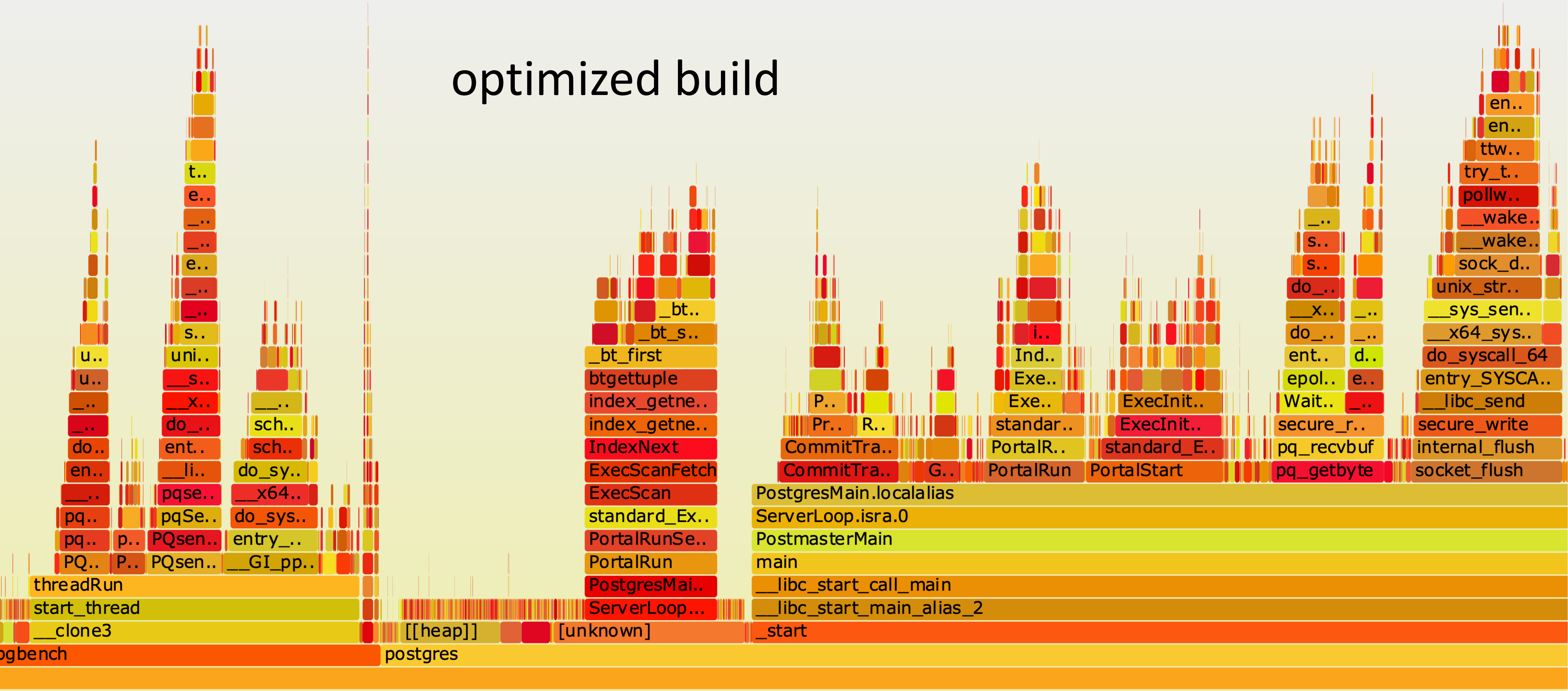
- Is workload storage or CPU bound?
- Does working set fit in shared buffers?
- Is workload read or write heavy or mixed?

if you only configure two things…

Postgres build

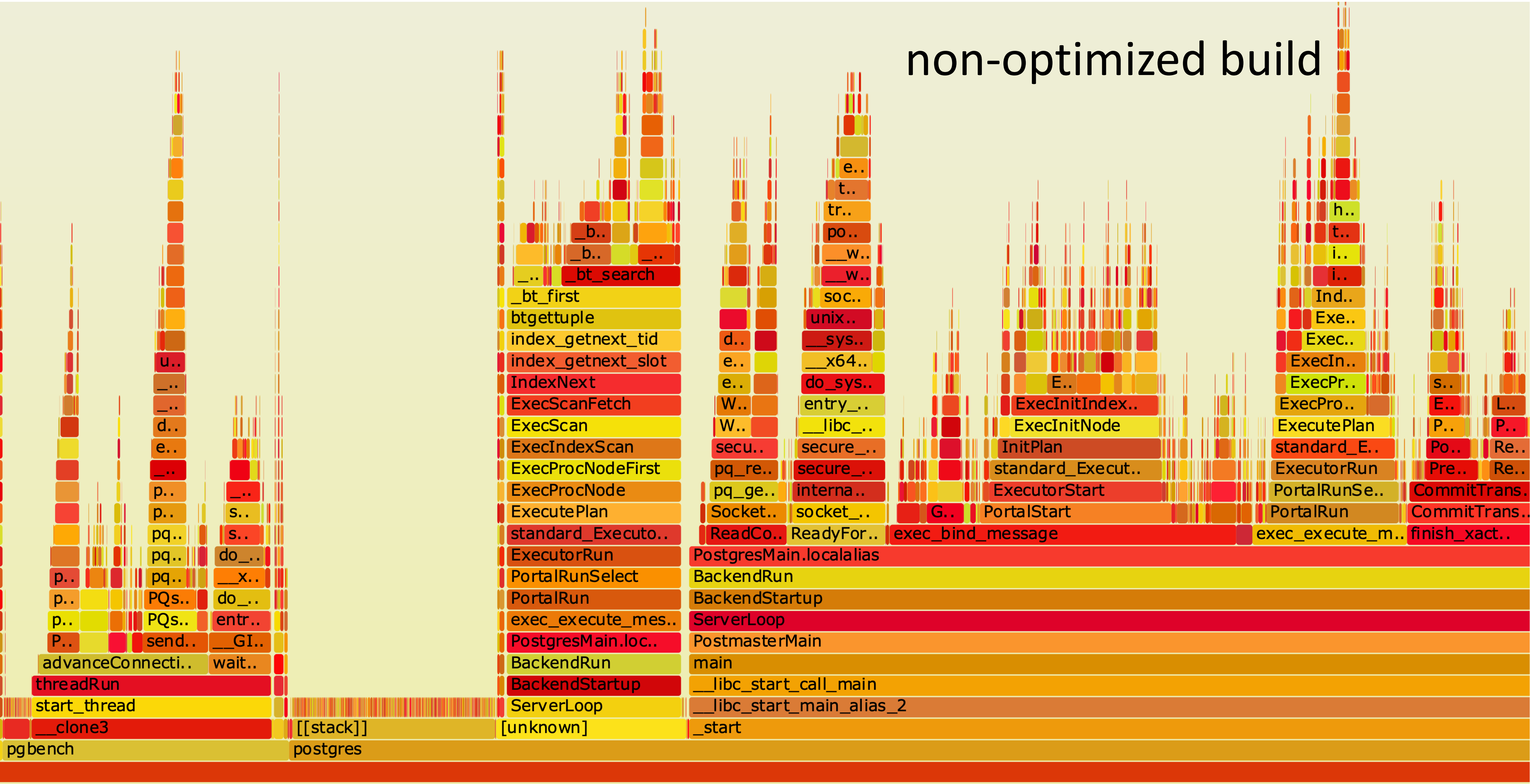
- applicable to all benchmarks
- non-assert
- non-debug
- optimization level > 0

optimized build



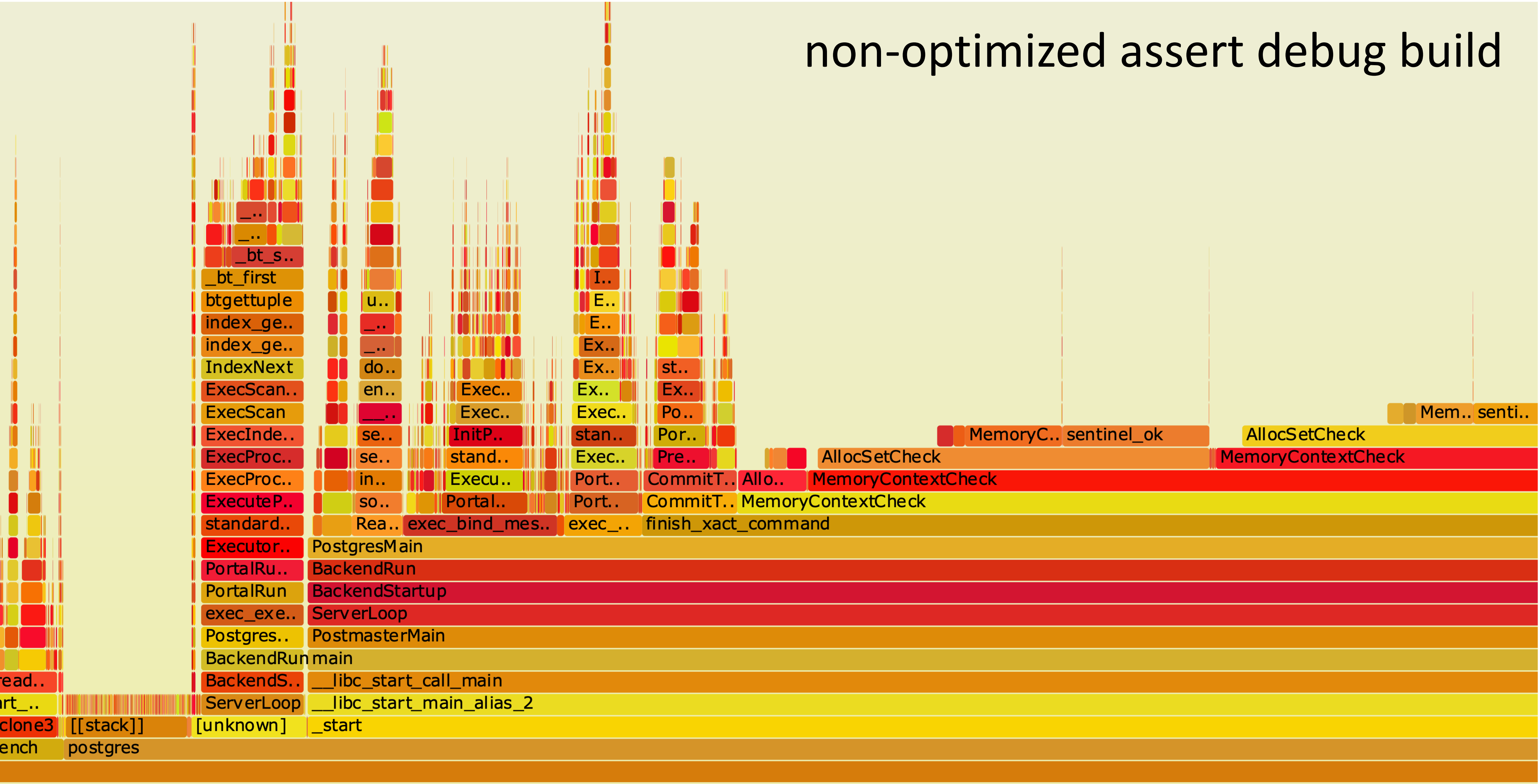
flame graphs courtesy of Andres Freund

non-optimized build



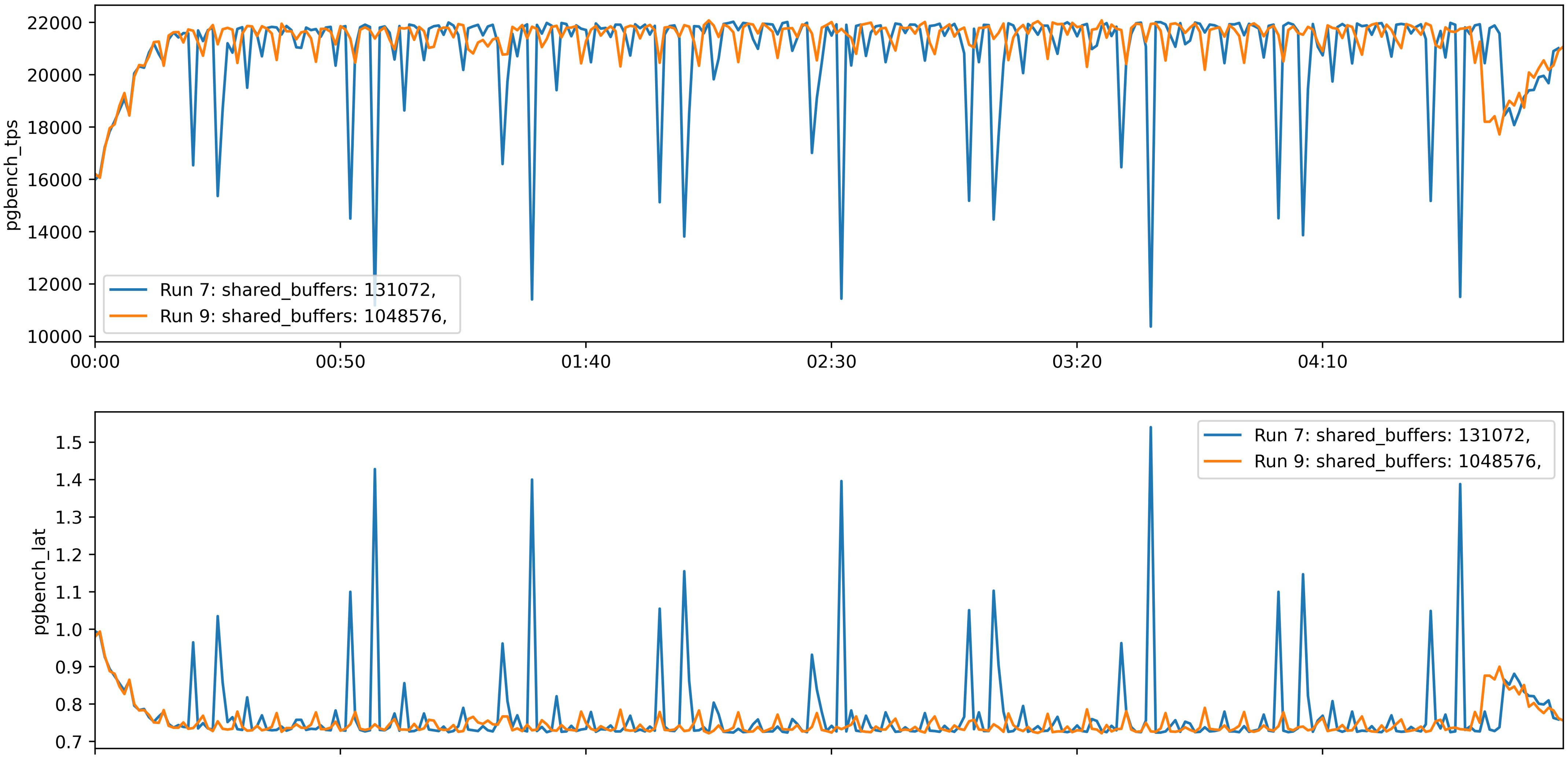
flame graphs courtesy of Andres Freund

non-optimized assert debug build



flame graphs courtesy of Andres Freund

postgres shared buffers

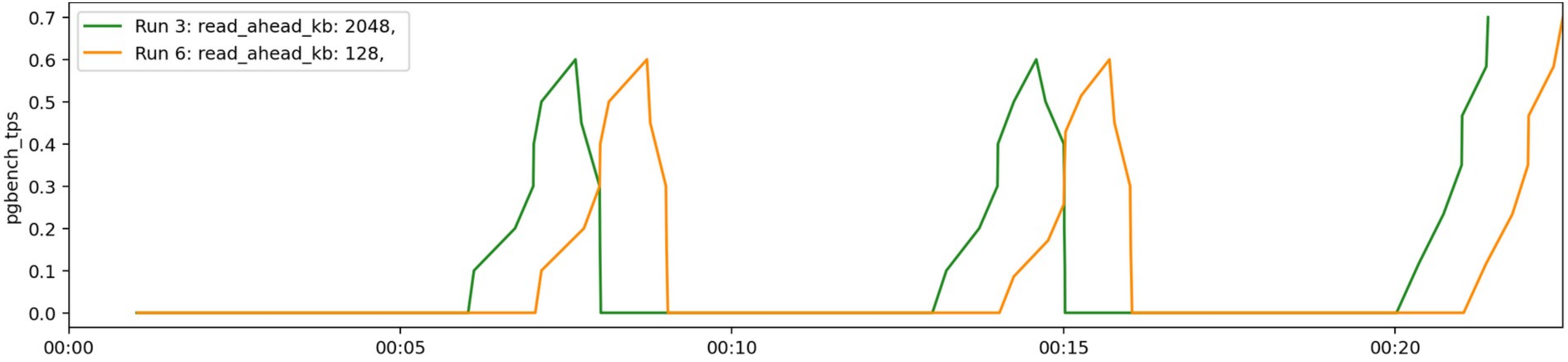


OS configuration matters when it controls your workload’s bottleneck

read_ahead_kb

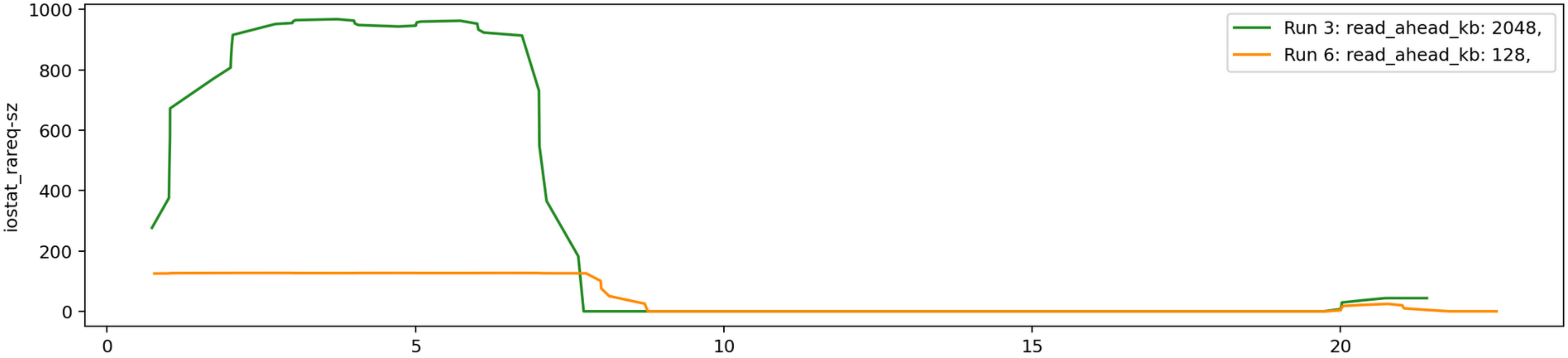
target readahead = sequential BW * latency

Larger read_ahead_kb finishes slightly sooner

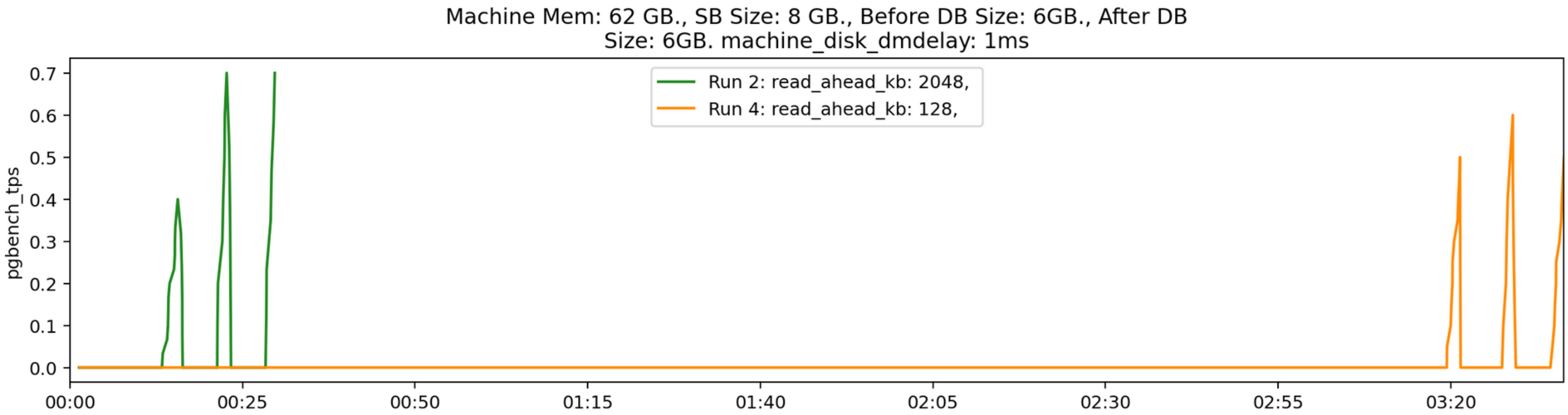


pgbench, SELECT * FROM large_table
5 GB table
1 client
3 transactions
8 GB shared buffers

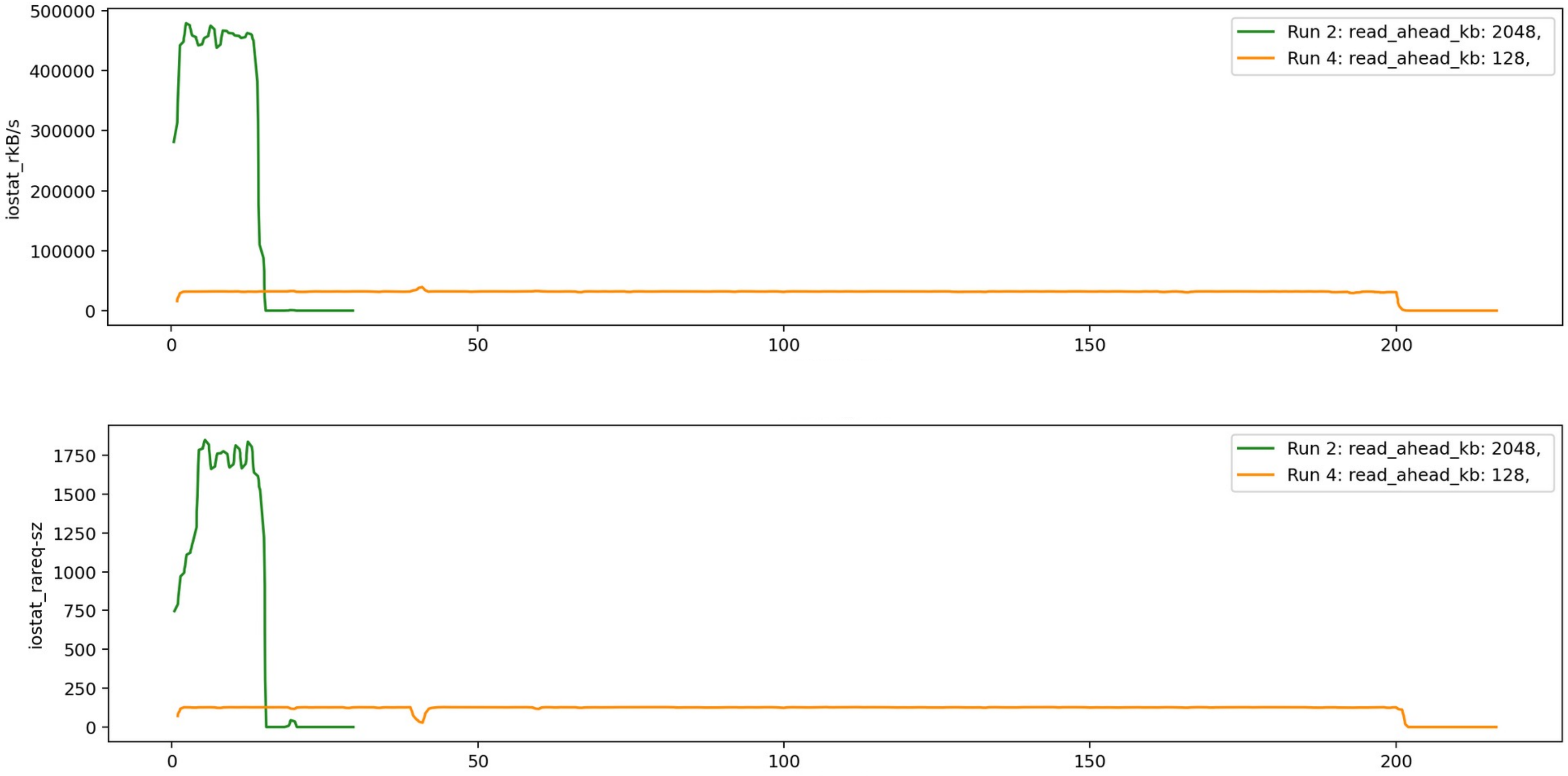
Read request size is much larger



With 1ms added latency via dmsetup delay, run with read_ahead_kb 2048 finishes in 30 seconds



Large request size and large read throughput



next steps

- Multiple workloads to simulate more complex scenarios
- Combining metrics from other sources

resources

- PgCon Ottawa 2023 benchmarking talk
  - https://speakerdeck.com/melanieplageman/o-performance-for-development
  - https://www.youtube.com/watch?v=CxyPZHG5beI
- pgbench docs
  - https://www.postgresql.org/docs/devel/pgbench.html