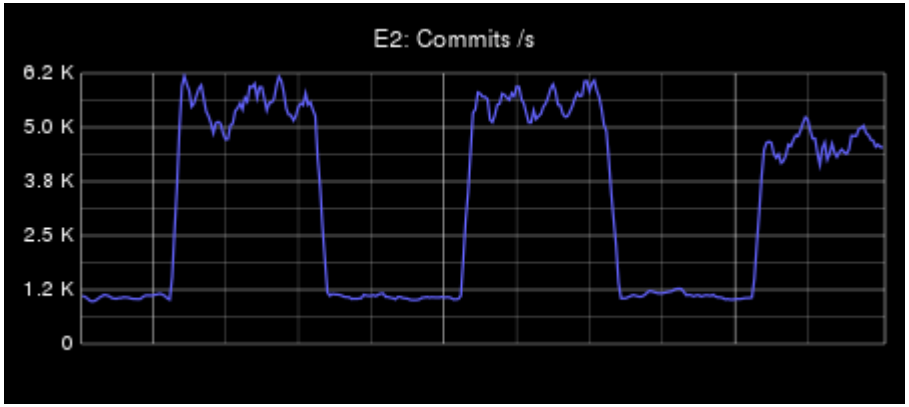
Booking.com

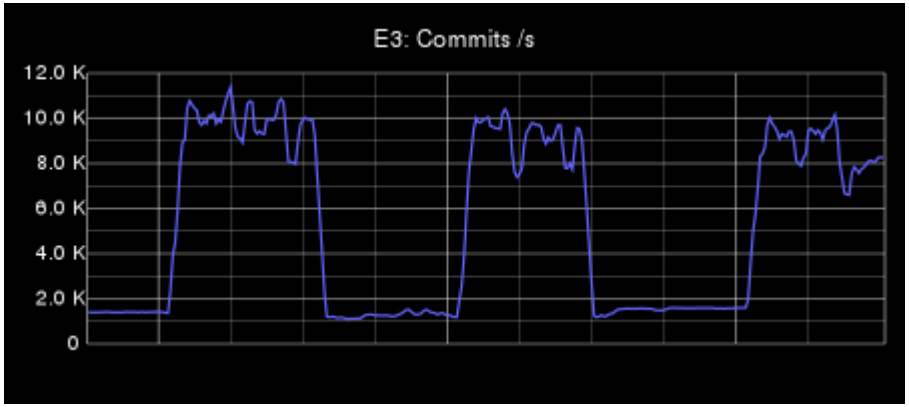
Parallel Replication in MySQL 5.7 and 8.0 by Booking.com

Presented at Pre-FOSDEM MySQL Day on Friday February 2nd, 2018

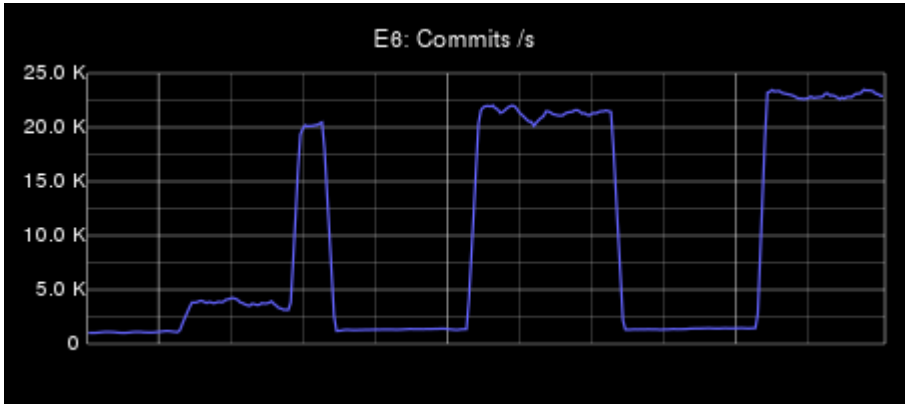
Eduardo Ortega (MySQL Database Engineer)
eduardo DTA ortega AT booking.com

Jean-François Gagné (System Engineer)
jeanfrancois DOT gagne AT booking.com


E2: Commits /s
6.2 K
5.0 K
3.8 K
2.5 K
1.2 K
0


E3: Commits /s
12.0 K
10.0 K
8.0 K
6.0 K
4.0 K
2.0 K
0


E5: Commits /s
12.5 K
10.0 K
7.5 K
5.0 K
2.5 K
0


E6: Commits /s
25.0 K
20.0 K
15.0 K
10.0 K
5.0 K
0

Booking.com

- Based in Amsterdam since 1996
- Online Hotel and Accommodation (Travel) Agent (OTA):
  - +1.636.000 properties in 229 countries
  - +1.555.000 room nights reserved daily
  - +40 languages (website and customer service)
  - +15.000 people working in 198 offices worldwide
- Part of the Priceline Group
- And we use MySQL:
  - Thousands (1000s) of servers

Booking.com’

- And we are hiring !
  - MySQL Engineer / DBA
  - System Administrator
  - System Engineer
  - Site Reliability Engineer
  - Developer / Designer
  - Technical Team Lead
  - Product Owner
  - Data Scientist
  - And many more…
- https://workingatbooking.com/

Session Summary

1. Introducing Parallel Replication (*// Replication*)
2. MySQL 5.7: Logical Clock and Intervals
3. MySQL 5.7: Tuning Intervals
4. Write Set in MySQL 8.0
5. Benchmark results from Booking.com with MySQL 8.0

// Replication

- Relatively new because it is hard
- It is hard because of data consistency
  - Running trx in // must give the same result on all slaves (= the master)
- Why is it important ?
  - Computers have many Cores, using a single one for writes is a waste
  - Some computer resources can give more throughput when used in parallel
    (RAID1 has 2 disks → we can do 2 Read IOs in parallel)
    (SSDs can serve many Read and/or Write IOs in parallel)

Reminder

- MySQL 5.6 has support for *schema* based parallel replication
- MySQL 5.7 adds support for *logical clock* parallel replication
  - In early version, the logical clock is group commit based
  - In current version, the logical clock is *interval* based
- MySQL 8.0 adds support for *Write Set* parallelism identification
- Write Set can also be found in MySQL 5.7 in Group replication

MySQL 5.7: LOGICAL CLOCK

- MySQL 5.7 has two `slave_parallel_type`:
  - both need "`SET GLOBAL slave_parallel_workers = N;`" (with N > 1)
  - `DATABASE`: the schema based // replication from 5.6 (not what we are talking about here)
  - `LOGICAL_CLOCK`: "Transactions that are part of the same binary log group commit on a master are applied in parallel on a slave." (from the doc. but not exact: Bug#85977)
  - the `LOGICAL_CLOCK` type is implemented by putting interval information in the binary logs
- `LOGICAL_CLOCK` is limited by the following:
  - Problems with long/big transactions
  - Problems with intermediate masters (*IM*)
- And it is optimized by slowing down the master to speedup the slave:
  - `binlog_group_commit_sync_delay`
  - `binlog_group_commit_sync_no_delay_count`

MySQL 5.7: LOGICAL CLOCK'

- Long transactions can block the parallel execution pipeline
- On the master: `---------- Time ---------->`

```
T1:  B------------------------C
T2:                          B--C
T3:                          B--C
```

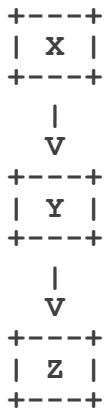
- On the slaves:

```
T1:  B------------------------C
T2:  B-- . . . . . . . . . . C
T3:  B-- . . . . . . . . . . C
```

- Try reducing as much as possible the number of big transactions:
  - Easier said than done: 10 ms is big compared to 1 ms
- Avoid monster transactions (LOAD DATA, unbounded UPDATE or DELETE, …)

MySQL 5.7: LOGICAL CLOCK”

- Replicating through intermediate masters (*IM*) shorten intervals
- Four transactions on X, Y and Z:



On X:

```
    ----Time---->
T1        B---C
T2        B---C
T3    B-------C
T4    B-------C
```

On Y:

```
    ----Time---->
T1      B---C
T2      B---C
T3      B-------C
T4      B-------C
```

On Z:

```
    ----Time--------->
T1    B---C
T2    B---C
T3          B-------C
T4          B-------C
```

- To get maximum replication speed, replace IM by Binlog Servers (or use MySQL 8.0)
- More details at http://blog.booking.com/better_parallel_replication_for_mysql.html

MySQL 5.7: LOGICAL CLOCK”’

- By default, MySQL 5.7 in logical clock does out-of-order commit:
  - There will be gaps (“`START SLAVE UNTIL SQL_AFTER_MTS_GAPS;`”)
  - Not replication crash safe without GTIDs
    http://jfg-mysql.blogspot.com/2016/01/replication-crash-safety-with-mts.html
  - And also be careful about these:
    binary logs content, `SHOW SLAVE STATUS`, skipping transactions, backups, …

- Using `slave_preserve_commit_order = 1` does what you expect:
  - This configuration does not generate gap
  - But it needs `log_slave_updates` (feature request to remove this limitation: Bug#75396)
  - Still it is not replication crash safe (surprising because no gap): Bug#80103 & Bug#81840
  - And it can hang if `slave_transaction_retries` is too low: Bug#89247

MySQL // Replication Guts: Intervals

- In MySQL (5.7 and higher), each transaction is tagged with two (2) numbers:
  - *sequence_number*: increasing id for each trx (not to confuse with GTID)
  - *last_committed*: sequence_number of the **latest trx** on which this trx depends (This can be understood as the “write view” of the current transaction)
- The `last_committed` / `sequence_number` pair is the parallelism *interval*
- Here an example of intervals for MySQL 5.7:

```
...
#170206 20:08:33 ... last_committed=6201 sequence_number=6203
#170206 20:08:33 ... last_committed=6203 sequence_number=6204
#170206 20:08:33 ... last_committed=6203 sequence_number=6205
#170206 20:08:33 ... last_committed=6203 sequence_number=6206
#170206 20:08:33 ... last_committed=6205 sequence_number=6207
...
```

MySQL 5.7 – Intervals Generation

MySQL 5.7 leverages parallelism on the master to generate intervals:

- `sequence_number` is an increasing id for each trx (not GTID)
  (Reset to 1 at the beginning of each new binary log)
- `last_committed` is (in MySQL 5.7) the sequence number of the most recently committed transaction when the current transaction gets its last lock
  (Reset to 0 at the beginning of each new binary log)

```
...
#170206 20:08:33 ... last_committed=6201 sequence_number=6203
#170206 20:08:33 ... last_committed=6203 sequence_number=6204
#170206 20:08:33 ... last_committed=6203 sequence_number=6205
#170206 20:08:33 ... last_committed=6203 sequence_number=6206
#170206 20:08:33 ... last_committed=6205 sequence_number=6207
...
```

MySQL – Intervals Quality

- For measuring parallelism identification quality with MySQL, we have a metric: the *Average Modified Interval Length* (*AMIL*)
- If we prefer to think in terms of group commit size, the AMIL can be mapped to a **pseudo**-group commit size by multiplying the AMIL by 2 and subtracting one
  - For a group commit of size n, the sum of the intervals length is n*(n+1) / 2

```
#170206 20:08:33 ... last_committed=6203 sequence_number=6204
#170206 20:08:33 ... last_committed=6203 sequence_number=6205
#170206 20:08:33 ... last_committed=6203 sequence_number=6206
```

MySQL – Intervals Quality

- For measuring parallelism identification quality with MySQL, we have a metric: the *Average Modified Interval Length* (*AMIL*)
- If we prefer to think in terms of group commit size, the AMIL can be mapped to a **pseudo**-group commit size by multiplying the AMIL by 2 and subtracting one
  - For a group commit of size `n`, the sum of the intervals length is `n*(n+1)/2`
  - ➢ `AMIL = (n+1)/2` (after dividing by n), algebra gives us `n = AMIL * 2 - 1`
- This mapping could give a hint for `slave_parallel_workers`

(http://jfg-mysql.blogspot.com/2017/02/metric-for-tuning-parallel-replication-mysql-5-7.html)

MySQL – Intervals Quality'

- Why do we need to "modify" the interval length ?
  - Because of a limitation in the current MTS applier which will only start trx 93136 once 93131 is completed → last_committed=93124 is modified to 93131

```
#170206 21:19:31 ... last_committed=93124 sequence_number=93131
#170206 21:19:31 ... last_committed=93131 sequence_number=93132
#170206 21:19:31 ... last_committed=93131 sequence_number=93133
#170206 21:19:31 ... last_committed=93131 sequence_number=93134
#170206 21:19:31 ... last_committed=93131 sequence_number=93135
#170206 21:19:31 ... last_committed=93124 sequence_number=93136
#170206 21:19:31 ... last_committed=93131 sequence_number=93137
#170206 21:19:31 ... last_committed=93131 sequence_number=93138
#170206 21:19:31 ... last_committed=93132 sequence_number=93139
#170206 21:19:31 ... last_committed=93138 sequence_number=93140
```

MySQL – Intervals Quality”

- Script to compute the Average Modified Interval Length:

```
file=my_binlog_index_file;
echo _first_binlog_to_analyse_ > $file;
mysqlbinlog --stop-never -R --host 127.0.0.1 $(cat $file) |
  grep "^#" | grep -e last_committed -e "Rotate to" |
  awk -v file=$file -F "[ \t]*|=" '$11 == "last_committed" {
                                     if (length($2) == 7) {$2 = "0" $2;}
                                     if ($12 < max) {$12 = max;} else {max = $12;}
                                     print $1, $2, $14 - $12;}
                                   $10 == "Rotate"{print $12 > file; close(file); max=0;}' |
  awk -F " |:" '{my_h = $2 ":" $3 ":" $4;}
                NR == 1 {d=$1; h=my_h; n=0; sum=0; sum2=0;}
                d != $1 || h < my_h {print d, h, n, sum, sum2; d=$1; h=my_h;}
                {n++; sum += $5; sum2 += $5 * $5;}'
```

(https://jfg-mysql.blogspot.com/2017/02/metric-for-tuning-parallel-replication-mysql-5-7.html)

MySQL – Intervals Quality”’

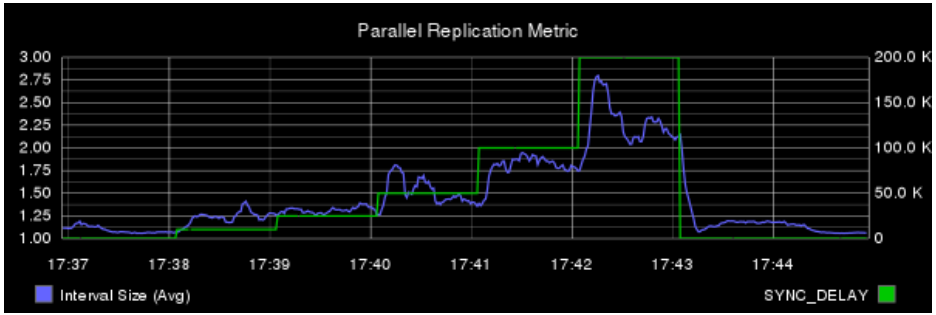
- Computing the AMIL needs parsing the binary logs
- This is complicated and needs to handle many special cases

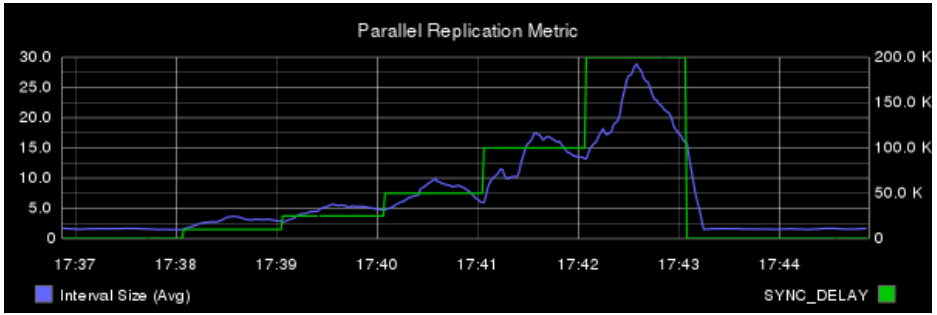
- Exposing counters for computing the AMIL would be better:
  - Bug#85965: Expose, on the master, counters for monitoring // information quality.
  - Bug#85966: Expose, on slaves, counters for monitoring // information quality.

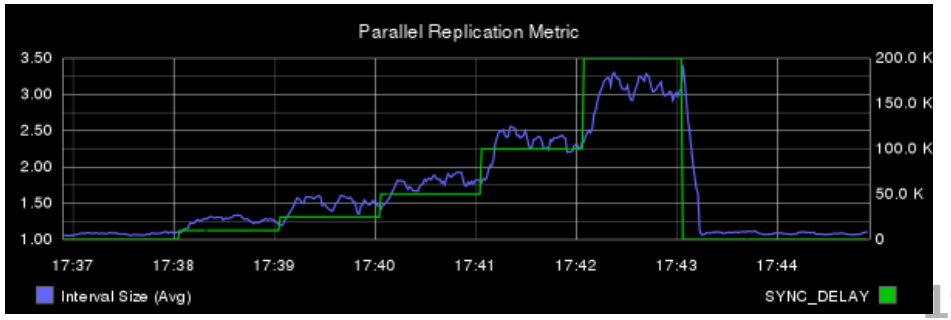
(https://jfg-mysql.blogspot.com/2017/02/metric-for-tuning-parallel-replication-mysql-5-7.html)

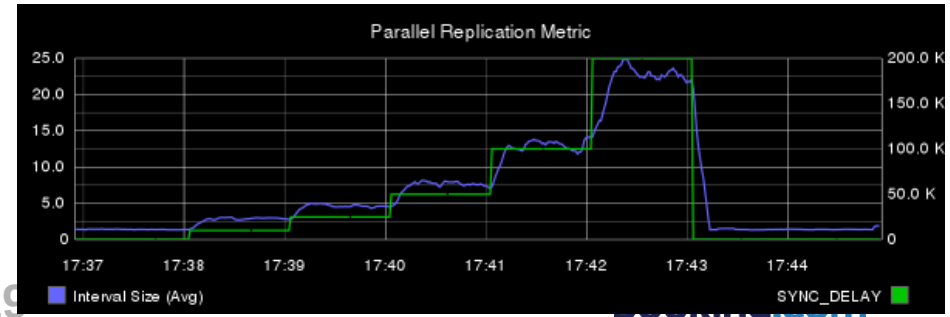
MySQL 5.7 – Tuning

- AMIL without and with tuning (delay) on four (4) Booking.com masters:
  (speed-up the slaves by increasing `binlog_group_commit_sync_delay`)









MySQL 8.0 – Write Set

- MySQL 8.0.1 introduced a new way to identify parallelism
- Instead of setting `last_committed` to “the seq. number of the most recently committed transaction when the current trx gets its last lock”…
- MySQL 8.0.1 uses “the sequence number of the last transaction that updated the same rows as the current transaction”
- To do that, MySQL 8.0 remembers which rows (tuples) are modified by each transaction: this is the *Write Set*
- Write Set are not put in the binary logs, they allow to “widen” the intervals

MySQL 8.0 – Write Set'

- MySQL 8.0.1 introduces new global variables to control Write Set:
  - `transaction_write_set_extraction = [ OFF | XXHASH64 ]`
  - `binlog_transaction_dependency_history_size` (default to 25000)
  - `binlog_transaction_dependency_tracking = [ COMMIT_ORDER | WRITESET_SESSION | WRITESET ]`

- `WRITESET_SESSION`: no two updates from the same session can be reordered
- `WRITESET`: any transactions which write different tuples can be parallelized

- `WRITESET_SESSION` will not work well for cnx recycling (Cnx Pools or Proxies):
  - Recycling a connection with `WRITESET_SESSION` impedes parallelism identification
  - Unless using the function reset_connection (with Bug#86063 fixed in 8.0.4)

MySQL 8.0 – Write Set”

- To use Write Set on a Master:
  - `transaction_write_set_extraction = XXHASH64`
  - `binlog_transaction_dependency_tracking = [ WRITESET_SESSION | WRITESET ]`
    (if `WRITESET`, `slave_preserve_commit_order` can avoid temporary inconsistencies)
- To use Write Set on an Intermediate Master (even single-threaded):
  - `transaction_write_set_extraction = XXHASH64`
  - `binlog_transaction_dependency_tracking = WRITESET`
    (`slave_preserve_commit_order` can avoid temporary inconsistencies)
- To stop using Write Set:
  - `binlog_transaction_dependency_tracking = COMMIT_ORDER`
  - `transaction_write_set_extraction = OFF`

MySQL 8.0 – Write Set'''

- Result for *single-threaded* Booking.com Intermediate Master (before and after):

```
#170409  3:37:13  [...] last_committed=6695 sequence_number=6696 [...]
#170409  3:37:14  [...] last_committed=6696 sequence_number=6697 [...]
#170409  3:37:14  [...] last_committed=6697 sequence_number=6698 [...]
#170409  3:37:14  [...] last_committed=6698 sequence_number=6699 [...]
#170409  3:37:14  [...] last_committed=6699 sequence_number=6700 [...]
#170409  3:37:14  [...] last_committed=6700 sequence_number=6701 [...]
#170409  3:37:14  [...] last_committed=6700 sequence_number=6702 [...]
#170409  3:37:14  [...] last_committed=6700 sequence_number=6703 [...]
#170409  3:37:14  [...] last_committed=6700 sequence_number=6704 [...]
#170409  3:37:14  [...] last_committed=6704 sequence_number=6705 [...]
#170409  3:37:14  [...] last_committed=6700 sequence_number=6706 [...]
```

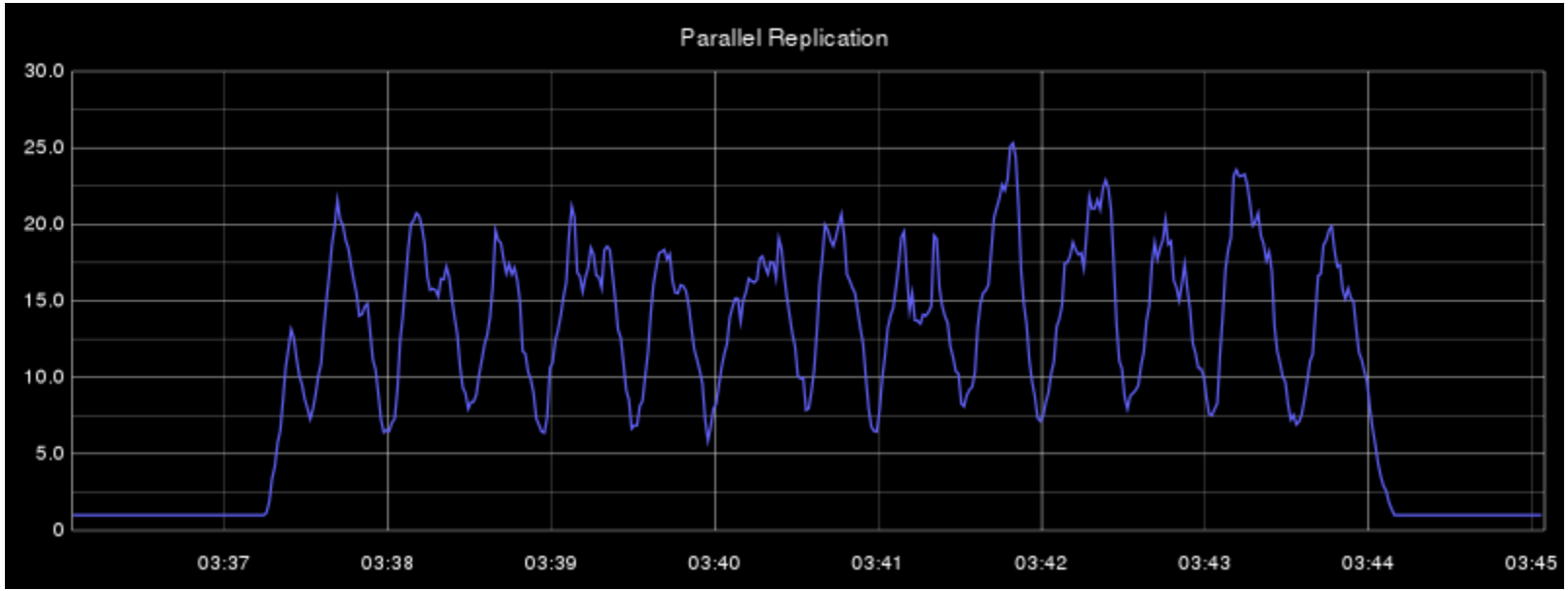
MySQL 8.0 – Write Set”’ ’

```
#170409  3:37:17  [...] last_committed=6700 sequence_number=6766 [...]
#170409  3:37:17  [...] last_committed=6752 sequence_number=6767 [...]
#170409  3:37:17  [...] last_committed=6753 sequence_number=6768 [...]
#170409  3:37:17  [...] last_committed=6700 sequence_number=6769 [...]
[...]
#170409  3:37:18  [...] last_committed=6700 sequence_number=6783 [...]
#170409  3:37:18  [...] last_committed=6768 sequence_number=6784 [...]
#170409  3:37:18  [...] last_committed=6784 sequence_number=6785 [...]
#170409  3:37:18  [...] last_committed=6785 sequence_number=6786 [...]
#170409  3:37:18  [...] last_committed=6785 sequence_number=6787 [...]
[...]
#170409  3:37:22  [...] last_committed=6785 sequence_number=6860 [...]
#170409  3:37:22  [...] last_committed=6842 sequence_number=6861 [...]
#170409  3:37:22  [...] last_committed=6843 sequence_number=6862 [...]
#170409  3:37:22  [...] last_committed=6785 sequence_number=6863
```

Booking.com

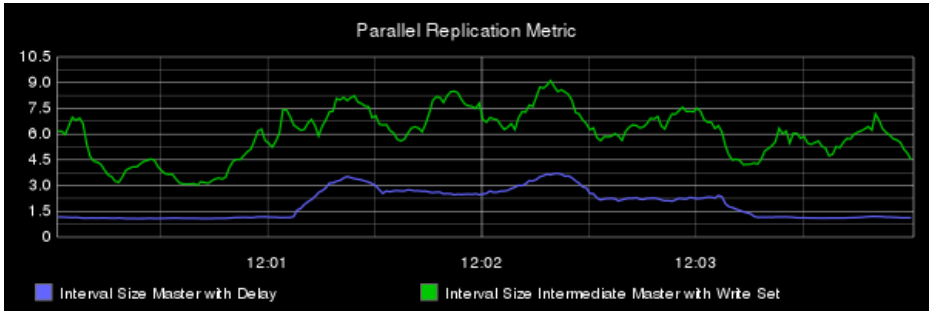
MySQL 8.0 – AMIL of Write Set

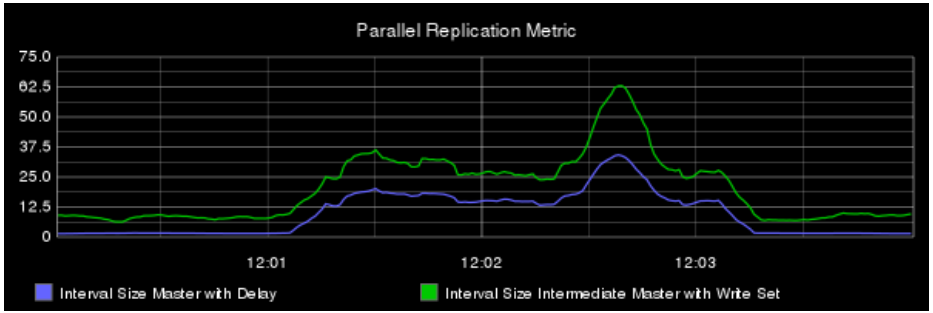
- AMIL on a single-threaded 8.0.1 Intermediate Master (*IM*) without/with Write Set:

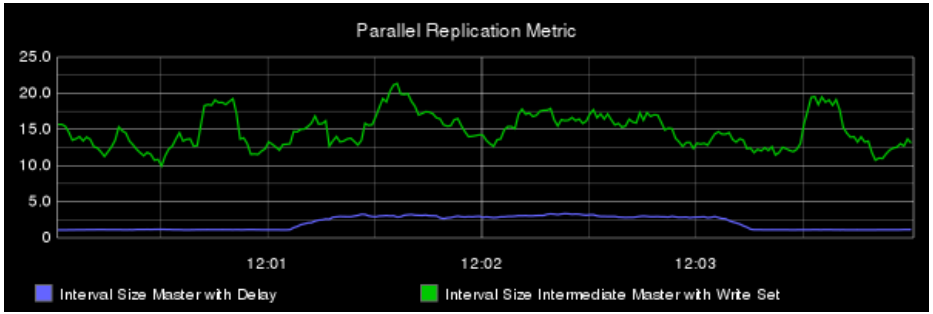


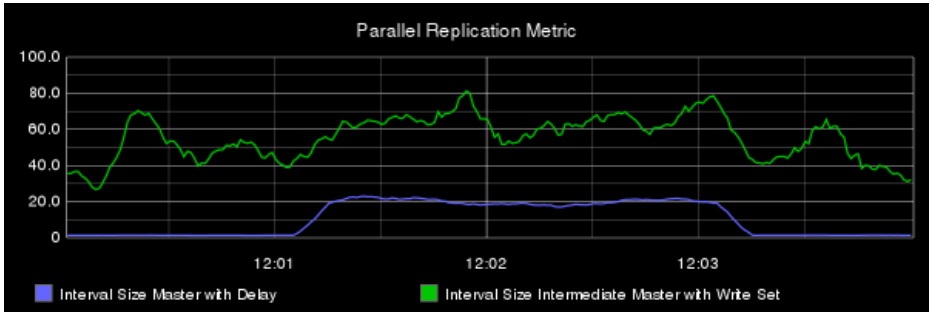
MySQL 8.0 – Write Set vs Delay

- AMIL on Booking.com masters with delay vs Write Set on Intermediate Master:









MySQL 8.0 – Write Set”’ ”

- Write Set advantages:
  - No need to slowdown the master (maybe not true in all cases)
  - Will work even at low concurrency on the master
  - Allows to test without upgrading the master (works on an intermediate master)
    (however, this sacrifices session consistency, which might give optimistic results)
  - Mitigate the problem of losing parallelism via intermediate masters
    (only with `binlog_transaction_dependency_tracking = WRITESET`)
    (→ the best solution is still Binlog Servers)

MySQL 8.0 – Write Set’’’ ’’’

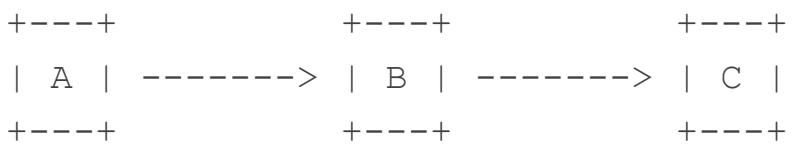
- Write Set limitations:
  - Needs Row-Based-Replication on the master (or intermediate master)
  - Not working for trx updating tables without PK and trx updating tables having FK (it will fall back to COMMIT_ORDER for those transactions)
  - Barrier at each DDL (Bug#86060 for adding counters)
  - Barrier at each binary log rotation: no transactions in different binlogs can be run in //
  - With `WRITESET_SESSION`, does not play well with connection recycling (Could use `COM_RESET_CONNECTION` if Bug#86063 is fixed)
- Write Set drawbacks:
  - Slowdown the master ? Consume more RAM ?
  - New technology: not fully mastered yet and there are bugs (still 1st DMR release)

MySQL 8.0 – Write Set @ B.com

- Tests on eight (8) real Booking.com environments (different workloads):
  - A is MySQL 5.6 and 5.7 masters (1 and 7), some are SBR (4) some are RBR (4)
  - B is MySQL 8.0.3 Intermediate Master with Write Set (RBR)

```
set global transaction_write_set_extraction = XXHASH64;
set global binlog_transaction_dependency_tracking = WRITESET;
```

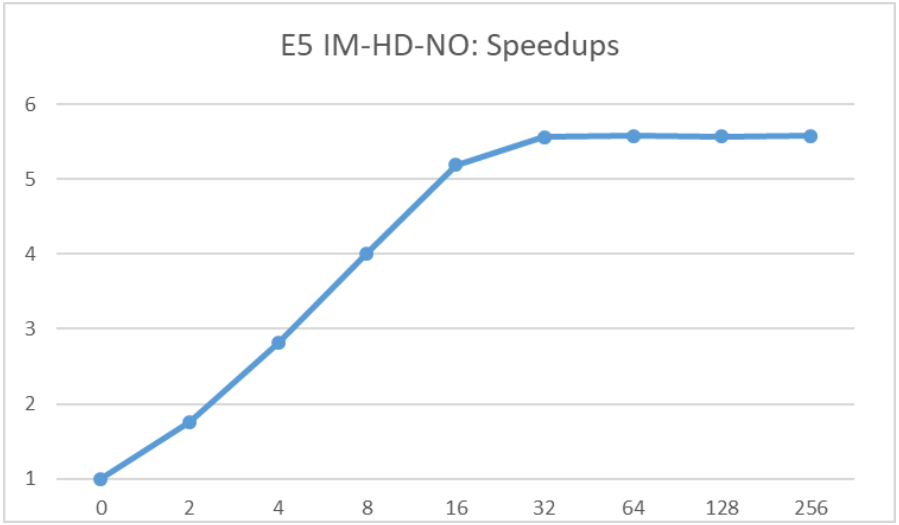
  - C is a slave with local SSD storage

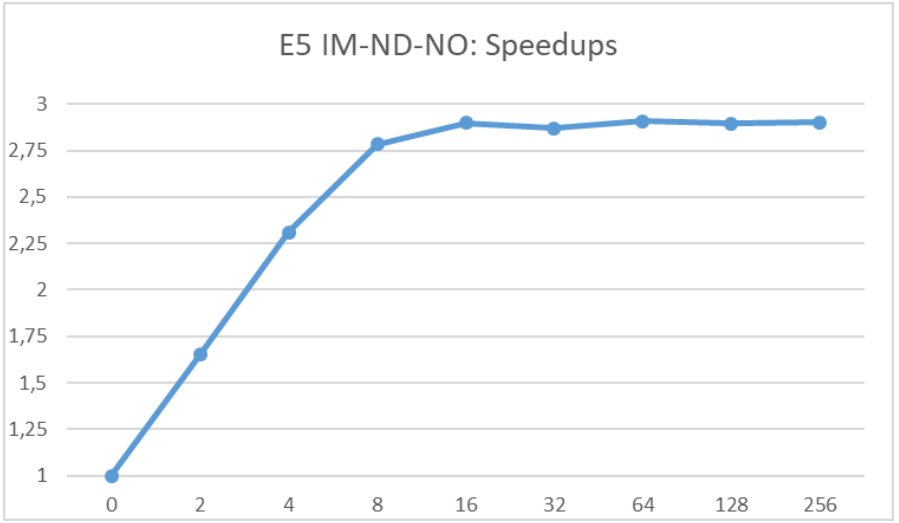


  - Run with 0, 2, 4, 8, 16, 32, 64, 128 and 256 workers,
    with High Durability (HD - sync_binlog = 1 & trx commit = 1) and No Durability (ND – 0 & 2),
    without and with `slave_preserve_commit_order` (NO and WO)
    with and without `log_slave_updates` (IM and SB)

Booking.com

MySQL 8.0 – Write Set Speedups

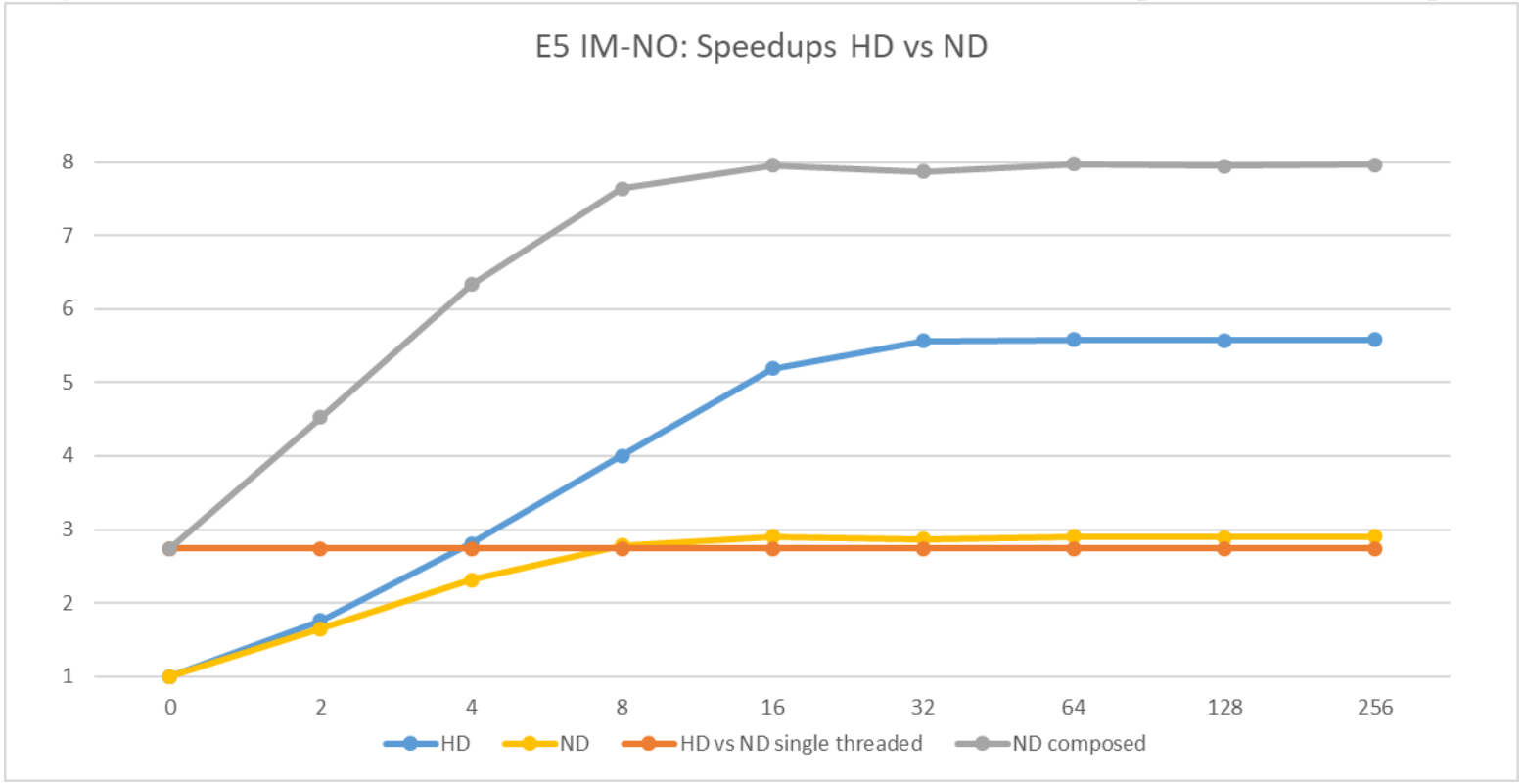




E5 IM-HD Single-Threaded: 6138 seconds

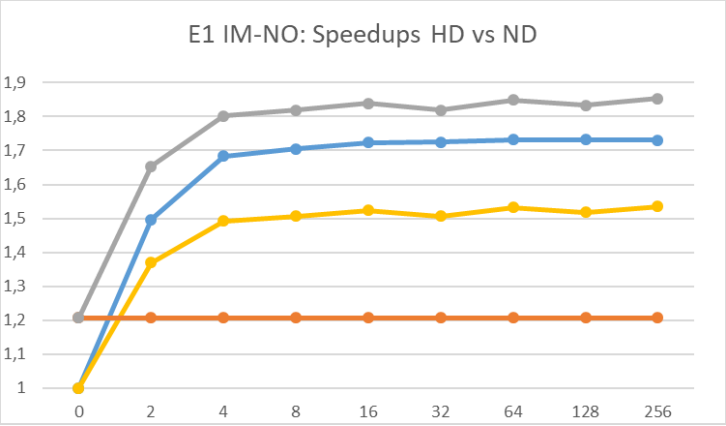
E5 IM-ND Single-Threaded: 2238 seconds

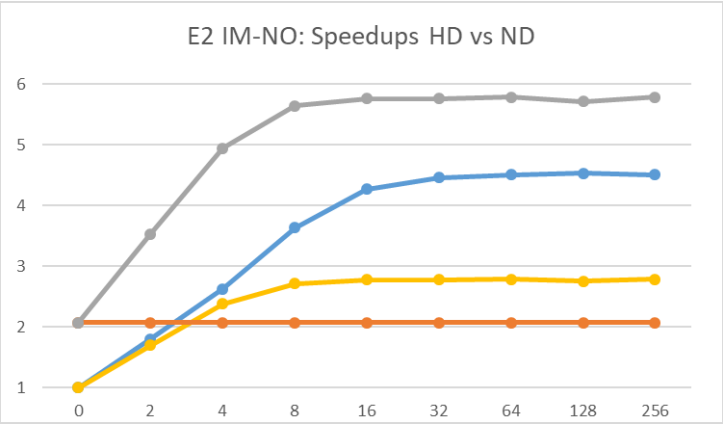
MySQL 8.0 – Write Set Speedups’

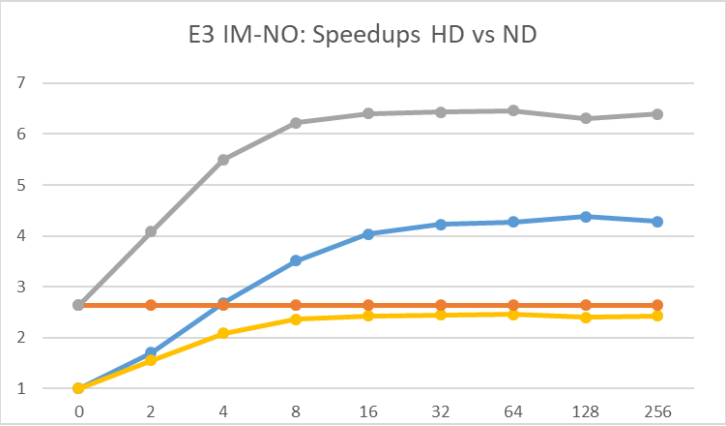


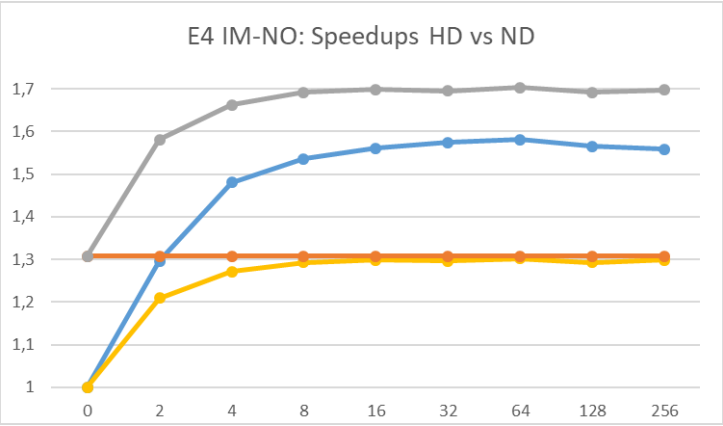
Booking.com

MySQL 8.0 – Write Set Speedups”



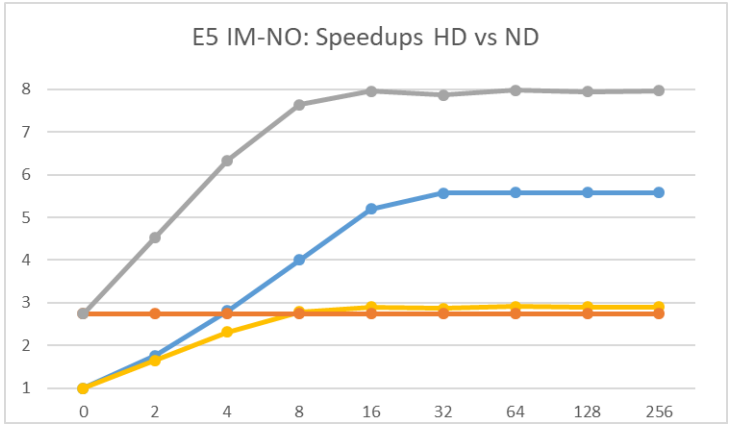


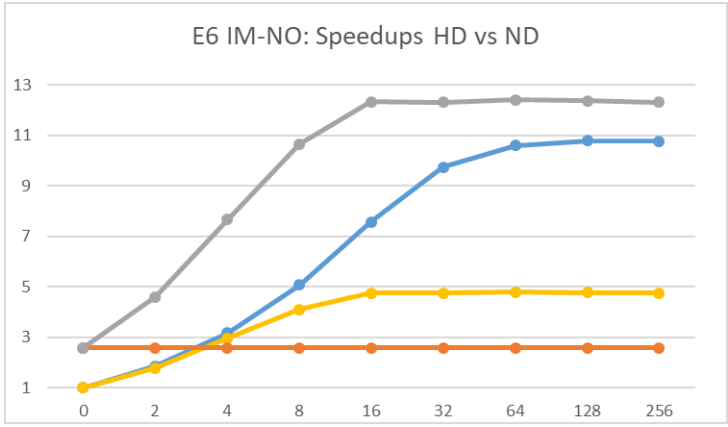


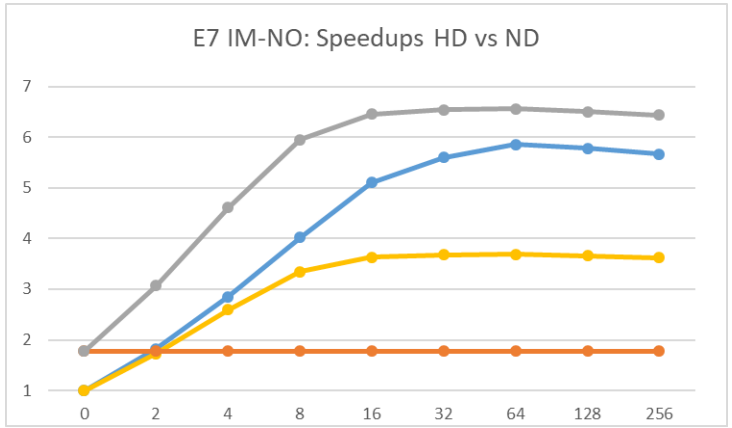


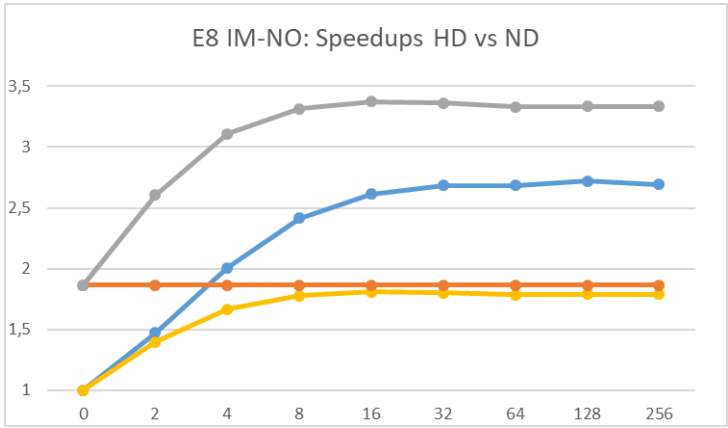
com

MySQL 8.0 – Write Set Speedups”’









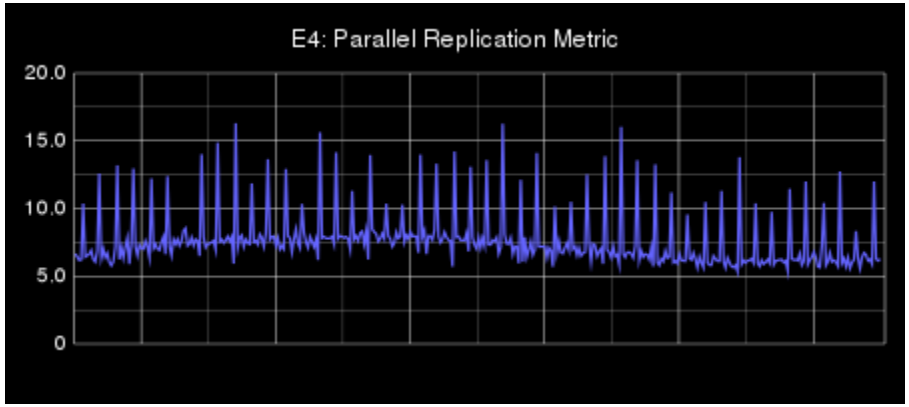
com

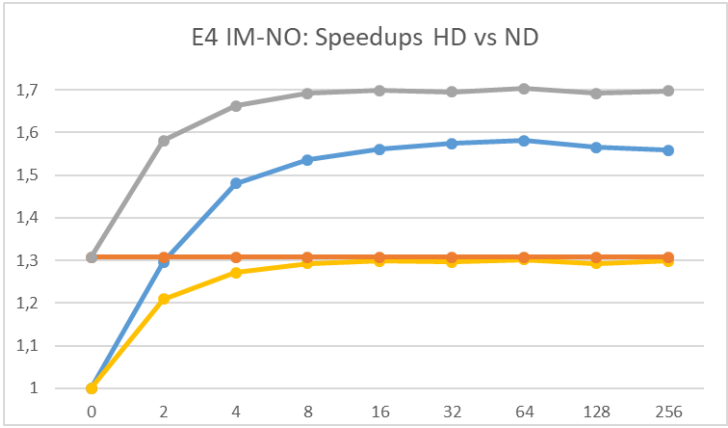
MySQL 8.0 – Speedup Summary

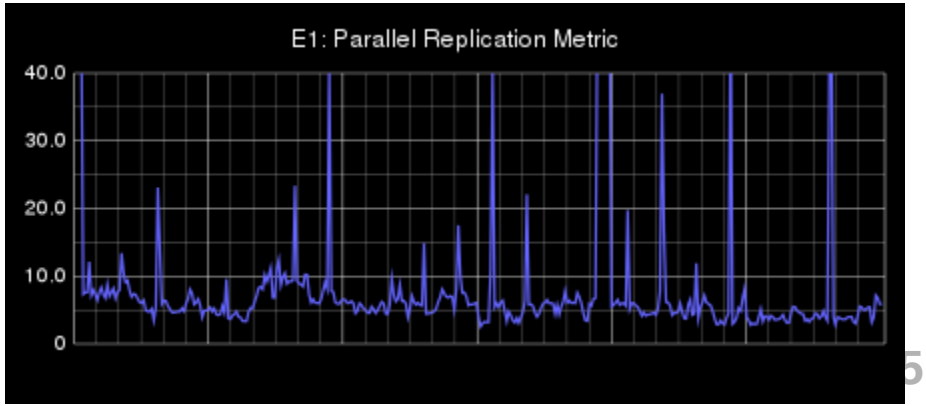
- No thrashing when too many threads !
- For the environments with High Durability:
  - Two (2) “interesting” speedups: 1.6, 1.7
  - One (1) good: 2.7
  - Four (4) very good speedups: 4.4, 4.5, 5.6, and 5.8
  - One (1) **great** speedups: 10.8 !
- For the environments without durability (ND):
  - Three (3) good speedups: 1.3, 1.5 and 1.8
  - Three (3) very good speedups: 2.4, 2.8 and 2.9
  - Two (2) **great** speedups: 3.7 and 4.8 !
- All that without tuning MySQL or the application

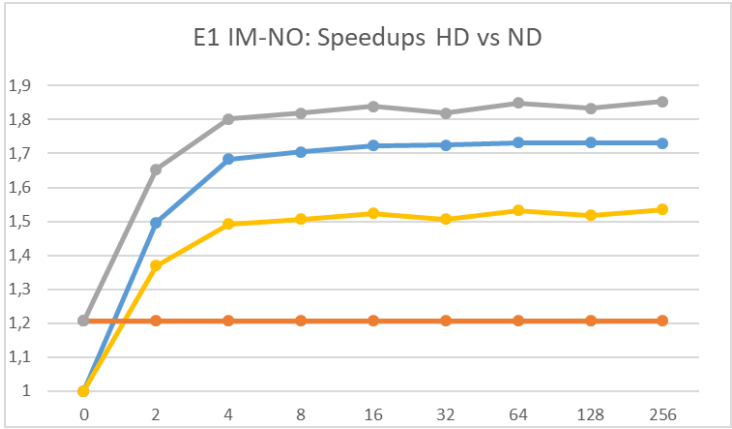
Booking.com

MySQL 8.0 – Looking at low speedups

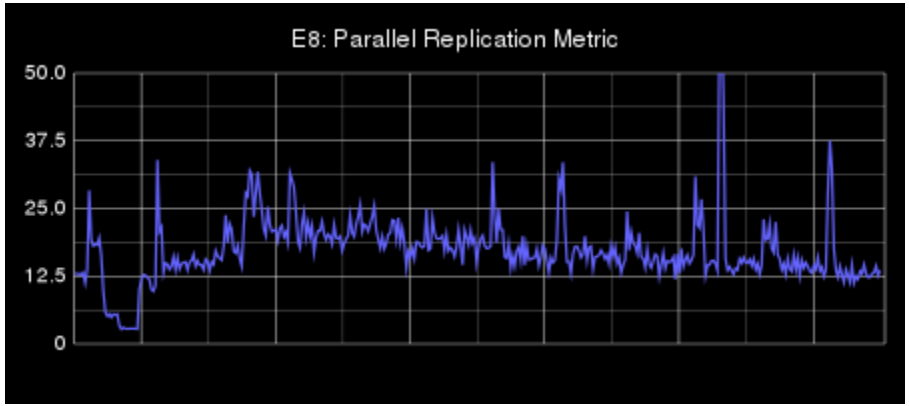


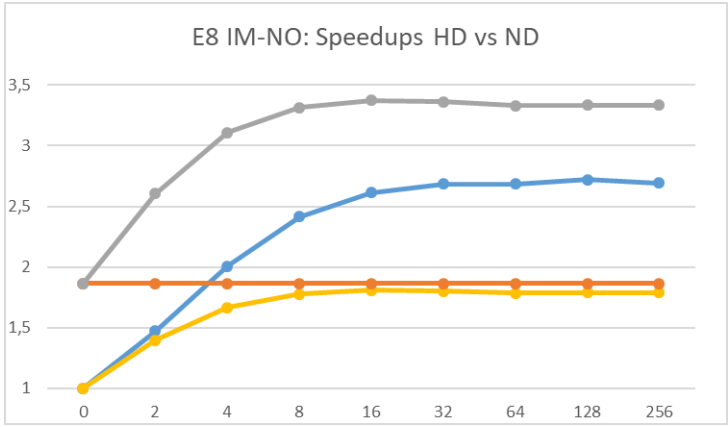






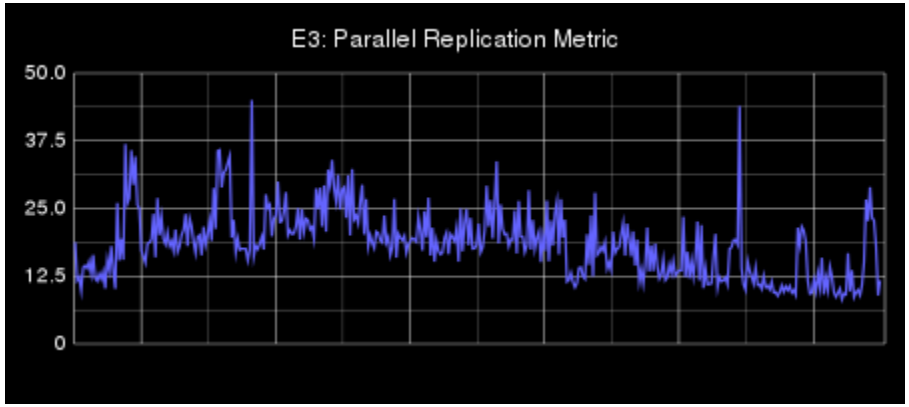
MySQL 8.0 – Looking at low speedups’

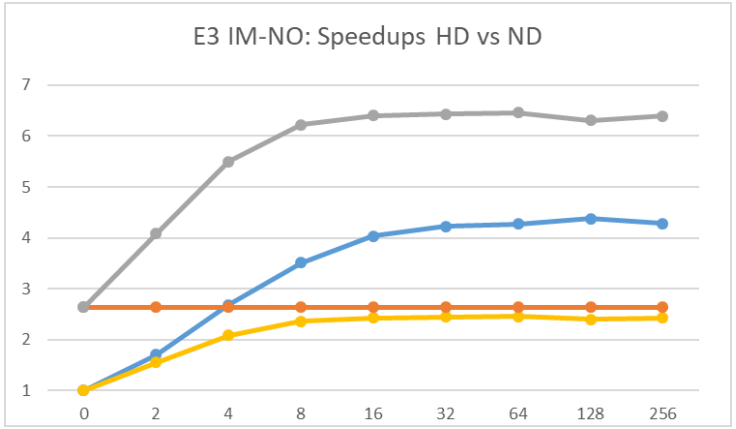


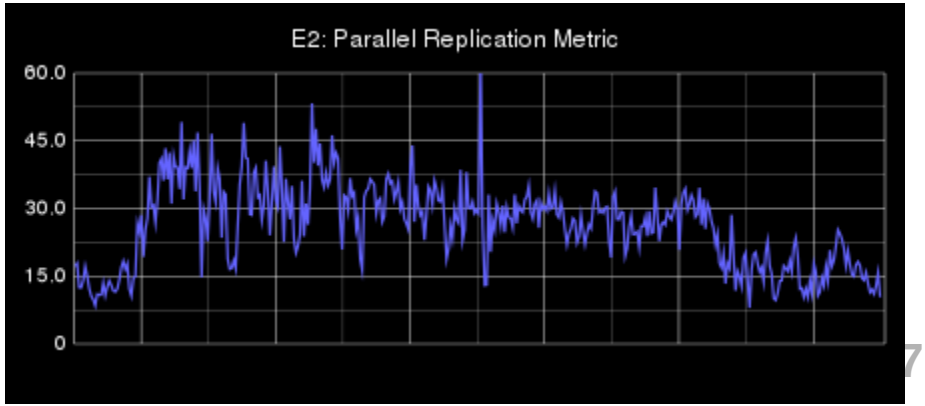


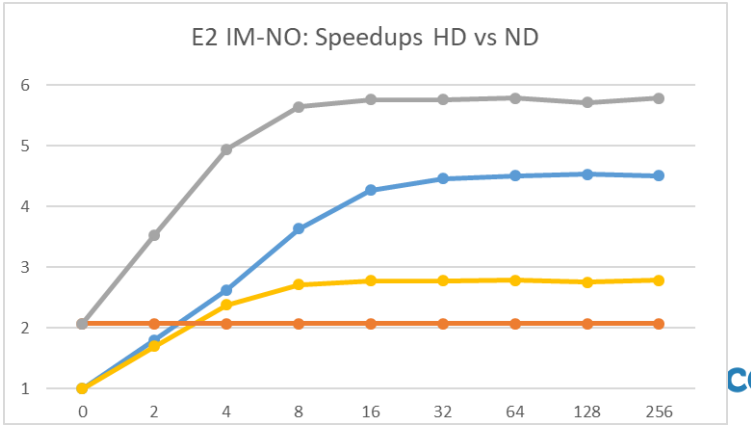
Booking.com

MySQL 8.0 – Looking at good speedups

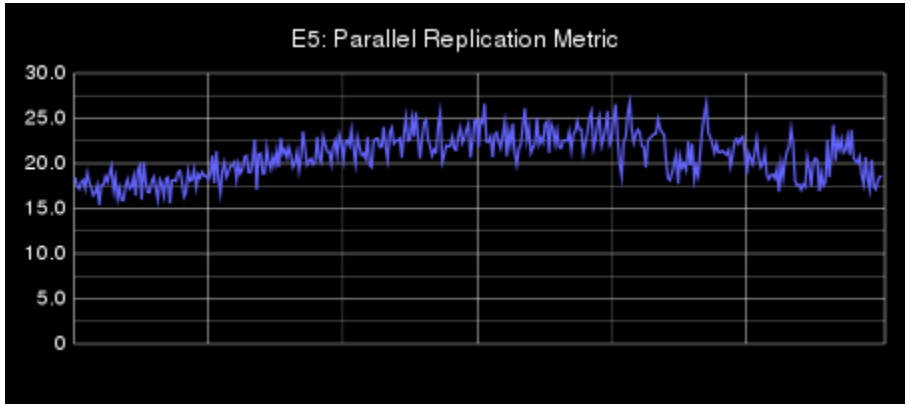


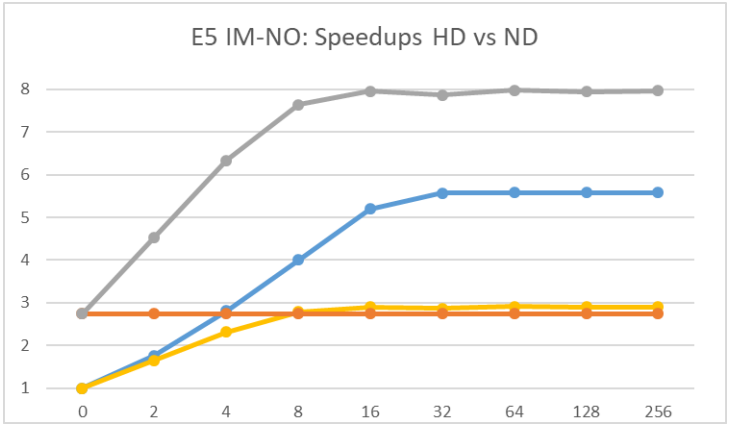


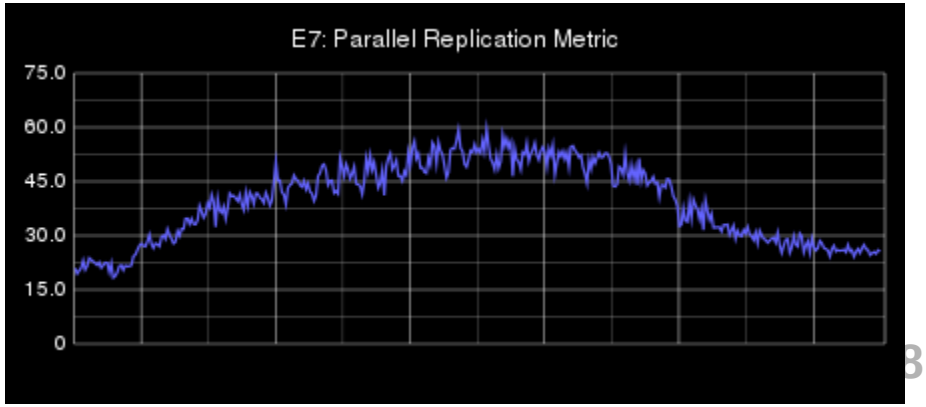


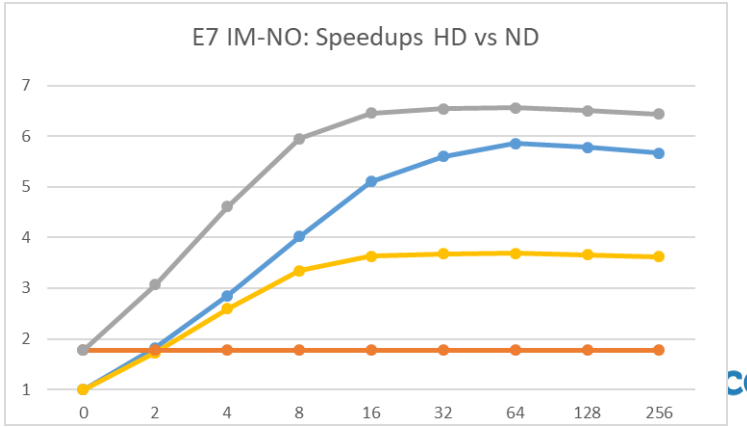


MySQL 8.0 – Looking at good speedups’

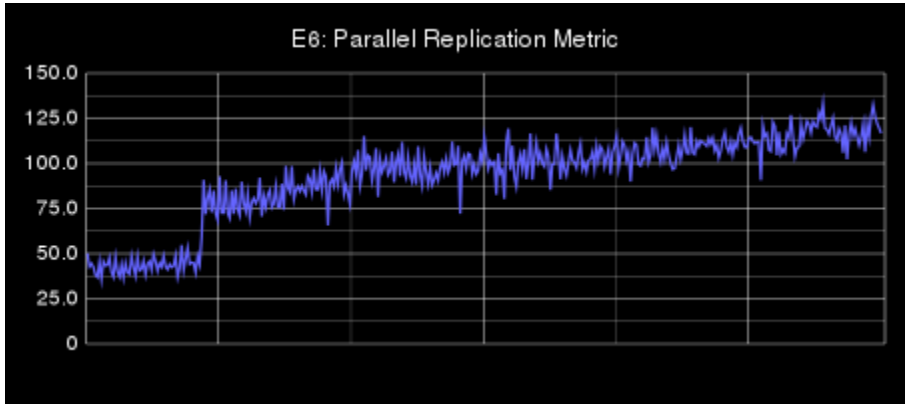


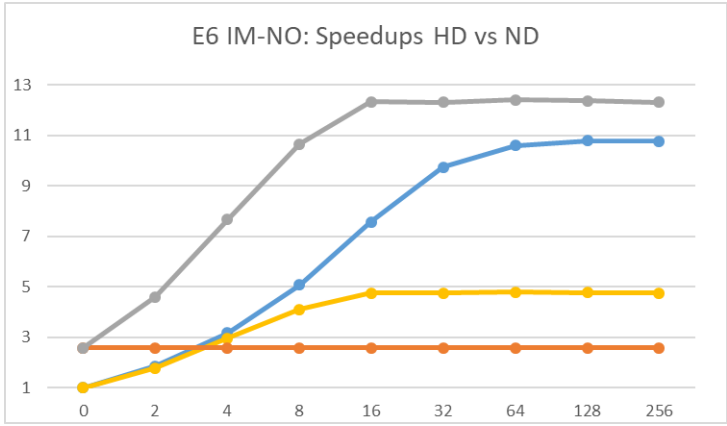






MySQL 8.0 – Looking at great speedups





Booking.com

How do we tune slave_parallel_workers?

Goal of tuning:
find a value that gives good speedup
without wasting too much resources

We can use:

- Commit rate
- Replication catch-up speed

But these are workload - dependent, making optimization harder:

- Workload variations over time
- Hardware differences

Wouldn’t it be cool to have a workload - independent metric to help optimize slave_parallel_workers?

Such a metric would:

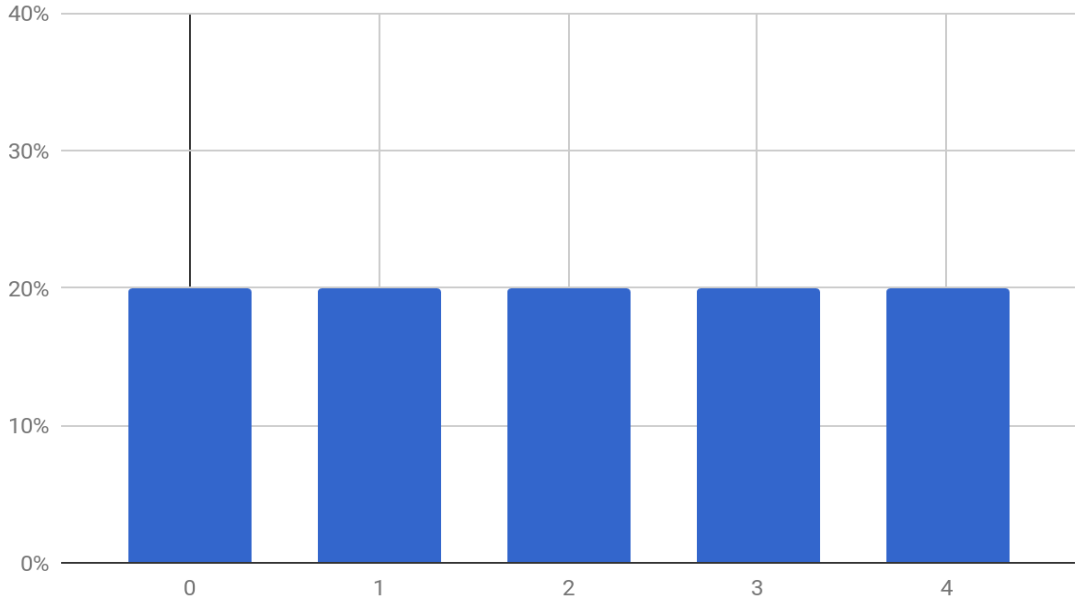
- Have a good value when all of your applier threads are doing work
- Be sensitive to having idle threads.

Sounds like a resource allocation fairness problem...

In a fair world...

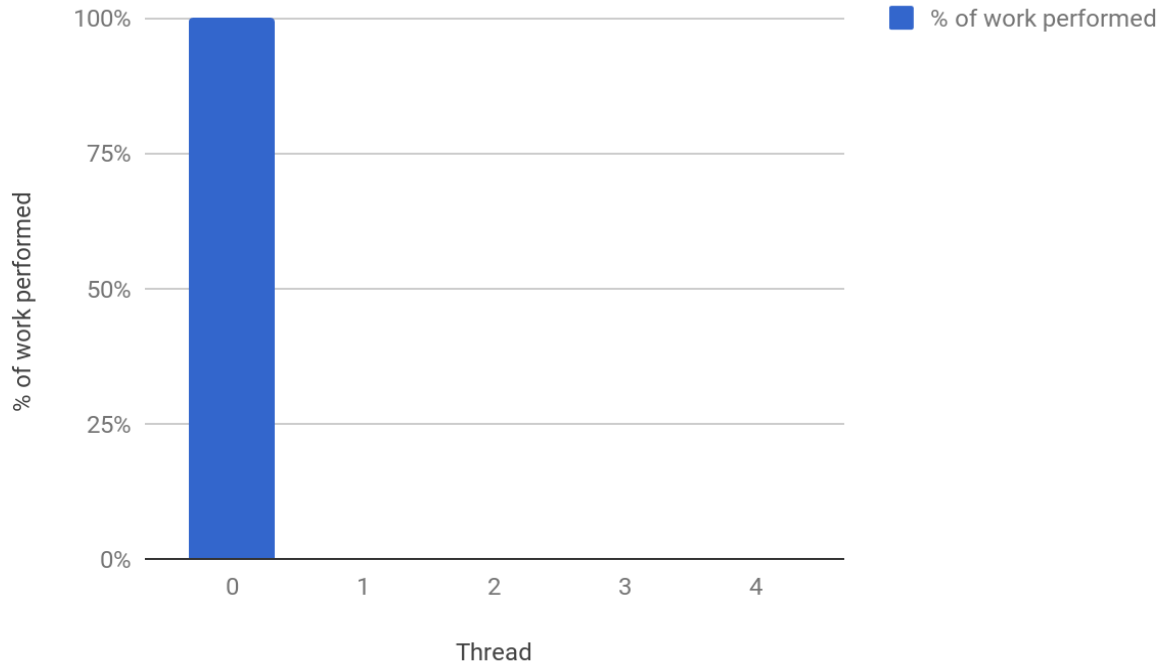
- slave_parallel_workers = *n*
- Each worker performs an equal amount of “work”

Say *n* = 5



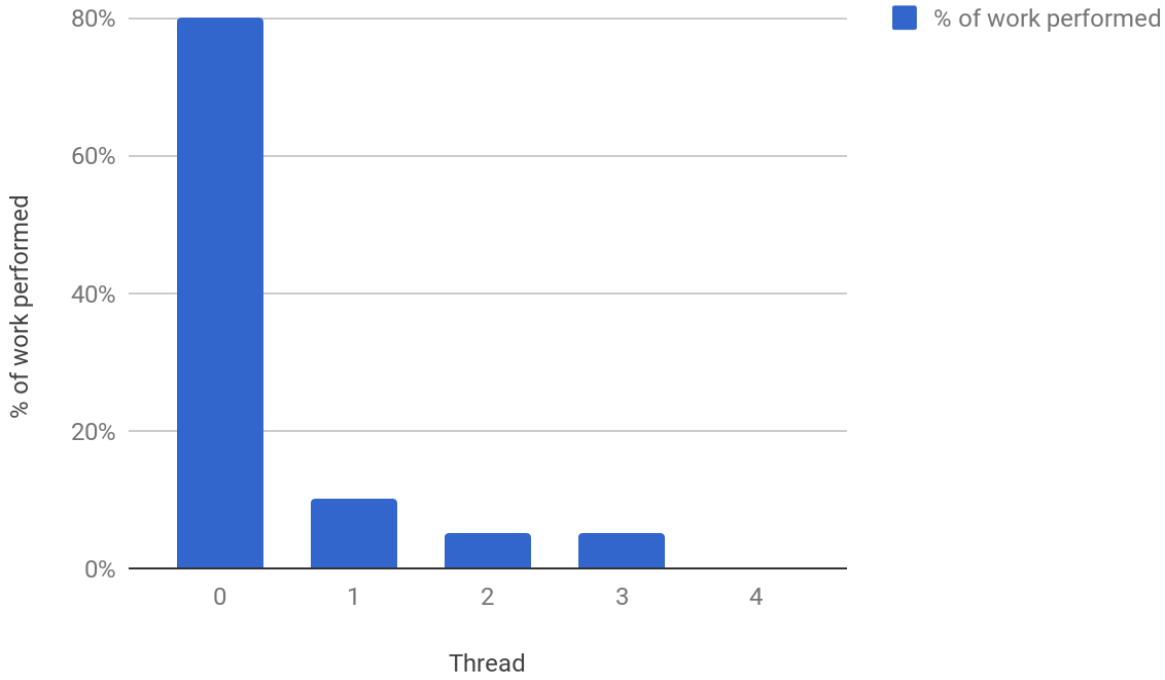
Stuff that would not be good:

- No actual parallelism



In an unfair world...

- Poor parallelism



How do we define “work”?

Inspired by Percona blog post by **Stephane Combaudon**

(https://www.percona.com/blog/2016/02/10/estimating-potential-for-mysql-5-7-parallel-replication/)

- Percentage of commits applied per worker thread

Easy to measure from performance schema:

```
USE performance_schema;

-- SET UP PS instrumentation
UPDATE setup_consumers SET enabled = 'YES' WHERE NAME LIKE 'events_transactions%';
UPDATE setup_instruments SET enabled = 'YES', timed = 'YES' WHERE NAME = 'transaction';

-- Get information on commits per applier thread
SELECT T.thread_id, T.count_star AS COUNT_STAR
  FROM events_transactions_summary_by_thread_by_event_name T
  WHERE T.thread_id IN (SELECT thread_id FROM replication_applier_status_by_worker);
```

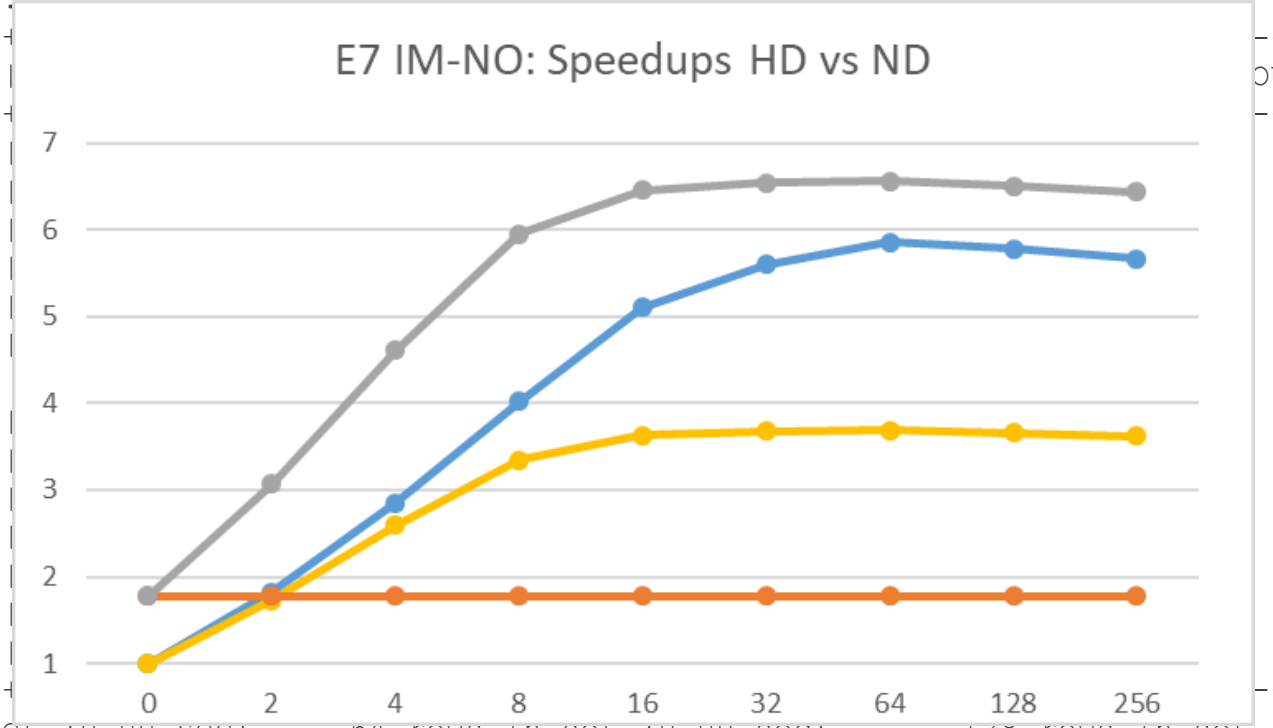
What does it give?

```
> SELECT [...];
+-----------+------------+
| thread_id | COUNT_STAR |
+-----------+------------+
|     63581 |        394 |
|     63582 |        292 |
|     63583 |        272 |
|     63584 |        260 |
|     63585 |        251 |
|     63586 |        237 |
[...]
|     63606 |        108 |
|     63607 |        106 |
|     63608 |        104 |
|     63609 |        100 |
|     63610 |         98 |
|     63611 |         94 |
|     63612 |         89 |
+-----------+------------+
32 rows in set (0.00 sec)
```

```
> SELECT [...];
+-----------+------------+
| thread_id | COUNT_STAR |
+-----------+------------+
|     63627 |        749 |
|     63628 |        504 |
|     63629 |        457 |
|     63630 |        433 |
|     63631 |        405 |
|     63632 |        384 |
[...]
|     63684 |         73 |
|     63685 |         73 |
|     63686 |         71 |
|     63687 |         68 |
|     63688 |         65 |
|     63689 |         65 |
|     63690 |         62 |
+-----------+------------+
64 rows in set (0.00 sec)
```

```
> SELECT [...];
+-----------+------------+
| thread_id | COUNT_STAR |
+-----------+------------+
|     63704 |        711 |
|     63705 |        468 |
|     63706 |        420 |
|     63707 |        378 |
|     63708 |        359 |
|     63709 |        340 |
[...]
|     63825 |          3 |
|     63826 |          3 |
|     63827 |          3 |
|     63828 |          3 |
|     63829 |          3 |
|     63830 |          2 |
|     63831 |          2 |
+-----------+------------+
128 rows in set (0.00 sec)
```

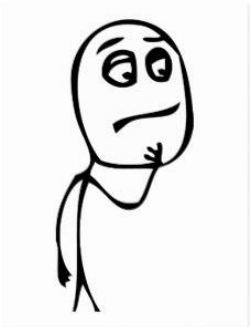
What does it give?




```
> SELECT [...]                    > SELECT [...]                    > SELECT [...]
+-----------+                                                                  ----------+
| thread_id |                                                      OUNT_STAR |
+-----------+                                                                  ----------+
|     63581 |                                                             711 |
|     63582 |                                                             468 |
|     63583 |                                                             420 |
|     63584 |                                                             378 |
|     63585 |                                                             359 |
|     63586 |                                                             340 |
[...]
|     63606 |                                                               3 |
|     63607 |                                                               3 |
|     63608 |                                                               3 |
|     63609 |                                                               3 |
|     63610 |                                                               3 |
|     63611 |                                                               2 |
|     63612 |                                                               2 |
+-----------+                                                                  ----------+
32 rows in set (0.00 sec)       64 rows in set (0.00 sec)       128 rows in set (0.00 sec)
```

How can we understand those numbers: Jain's index

```
> SELECT [...];
+-----------+------------+
| thread_id | COUNT_STAR |
+-----------+------------+
|     63627 |        749 |
|     63628 |        504 |
|     63629 |        457 |
|     63630 |        433 |
|     63631 |        405 |
|     63632 |        384 |
[...]
|     63684 |         73 |
|     63685 |         73 |
|     63686 |         71 |
|     63687 |         68 |
|     63688 |         65 |
|     63689 |         65 |
|     63690 |         62 |
+-----------+------------+
64 rows in set (0.00 sec)
```



Jain's fairness index:

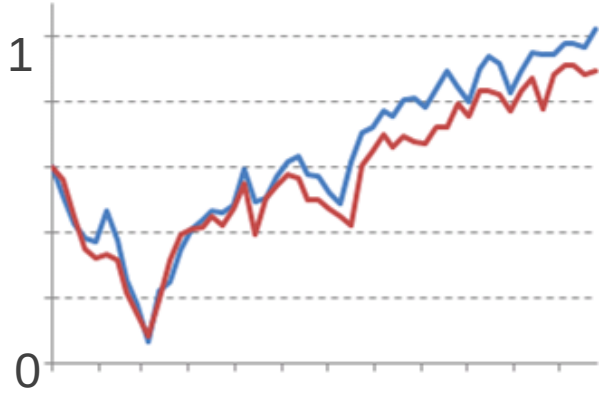
- n: number of applier threads.
- i: the i-th applier thread.
- x_i :amount of work performed by thread

$$\mathcal{J}(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} {x_i}^2}$$

What does Jain's index mean?

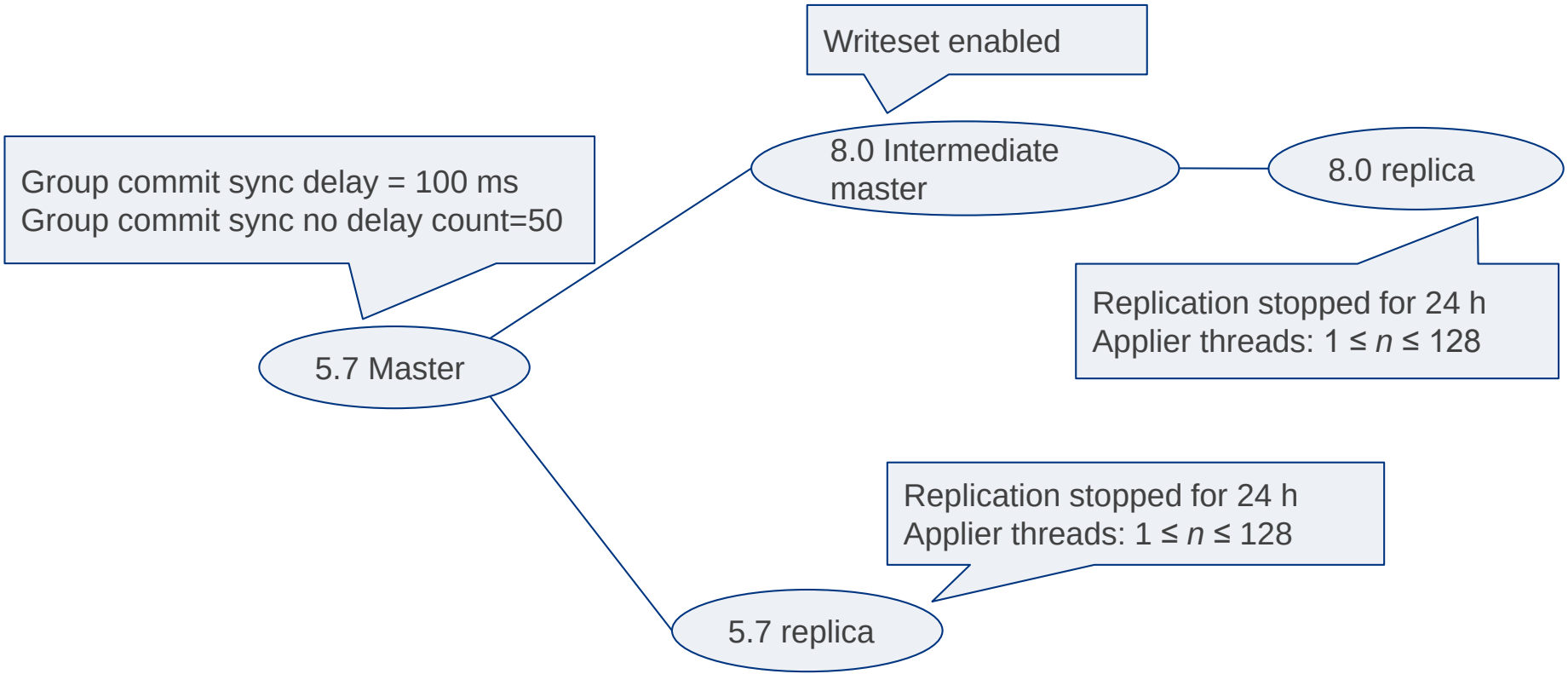
- $0 \leq J \leq 1$

- For $n = 1 \rightarrow J = 1$ (but no parallelism)
- If n > 1 and J ≅ 1, all workers are doing similar work
- If n > 1 and J < 1, some workers are doing less work
- If n > 1 and J << 1, some workers are idle

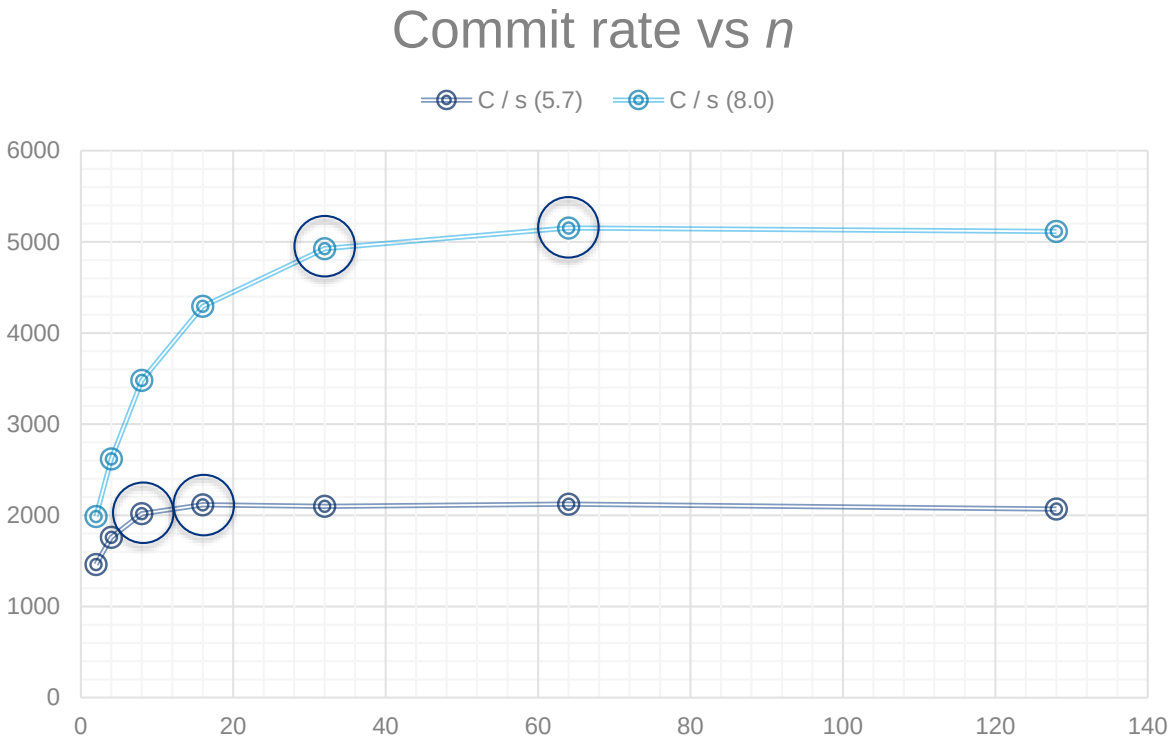


Avoid J ≅ 1
And avoid J << 1

Putting the idea to the test



First, something familiar to compare with:

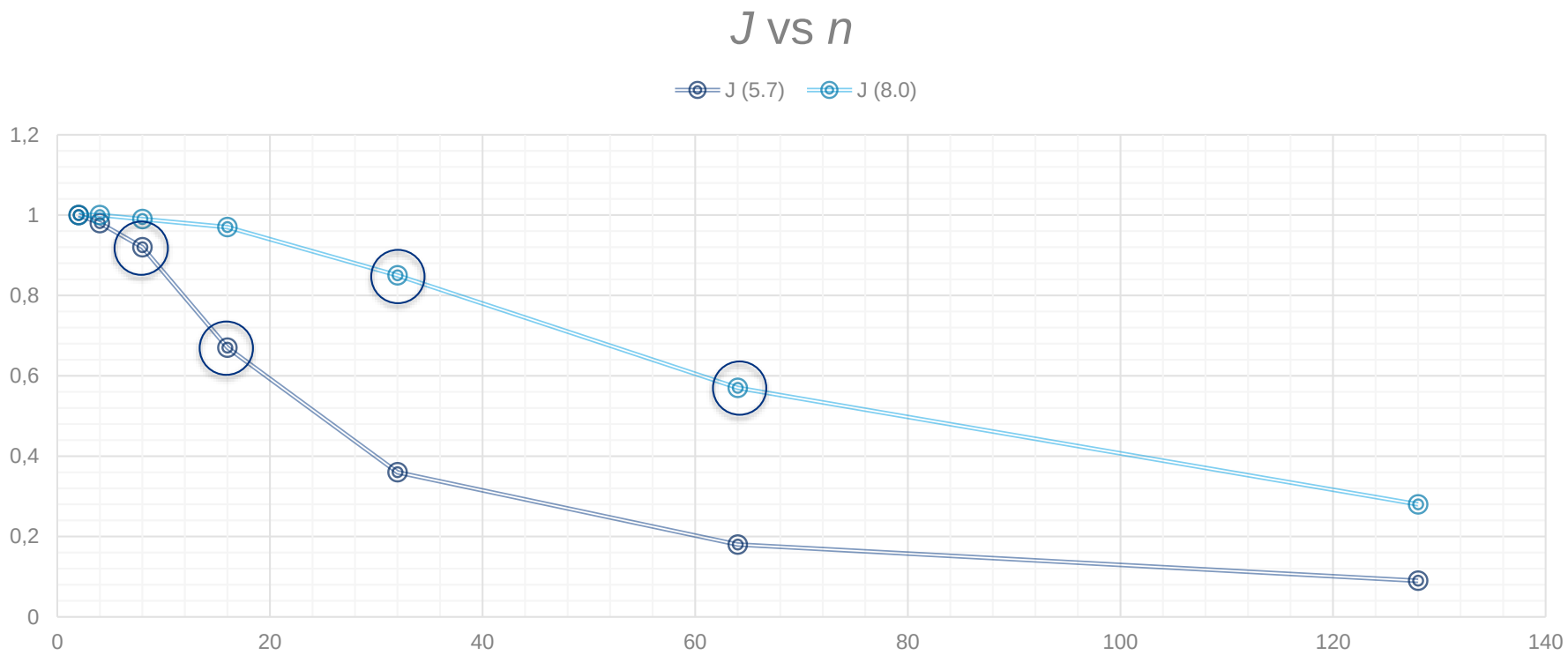


Optimal values for *n*:

- For 5.7, *n* = 8 or 16
- For 8.0, *n* = 32 or 64

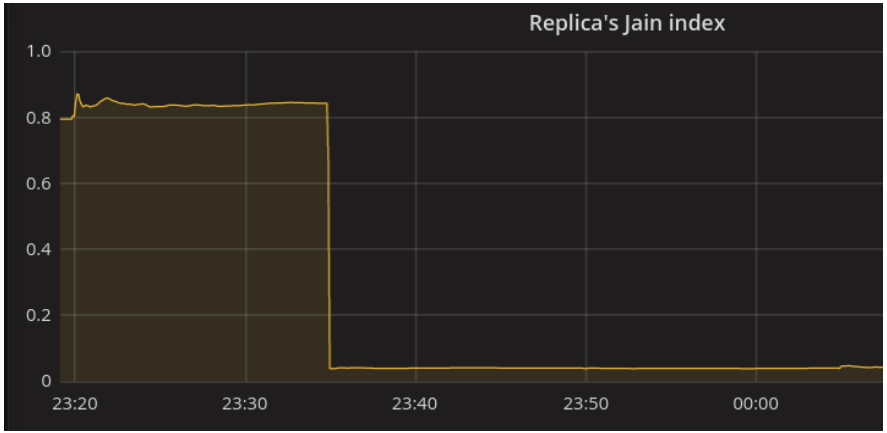
Higher values of *n* would be wasting resources on applier threads that have no work to do.

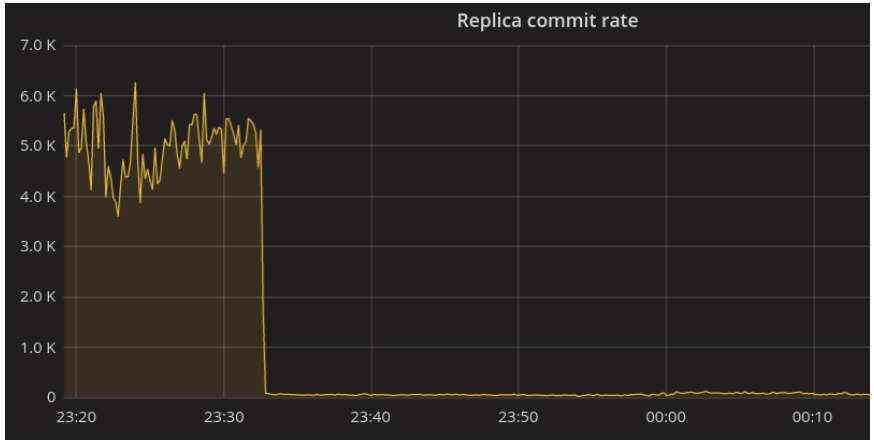
What does Jain´s index have to say?



Caveats...

- What happens when not catching up but just keeping up with current replication traffic?





Not enough work means idle threads

→ Optimal number of workers depends on catching up or keeping up

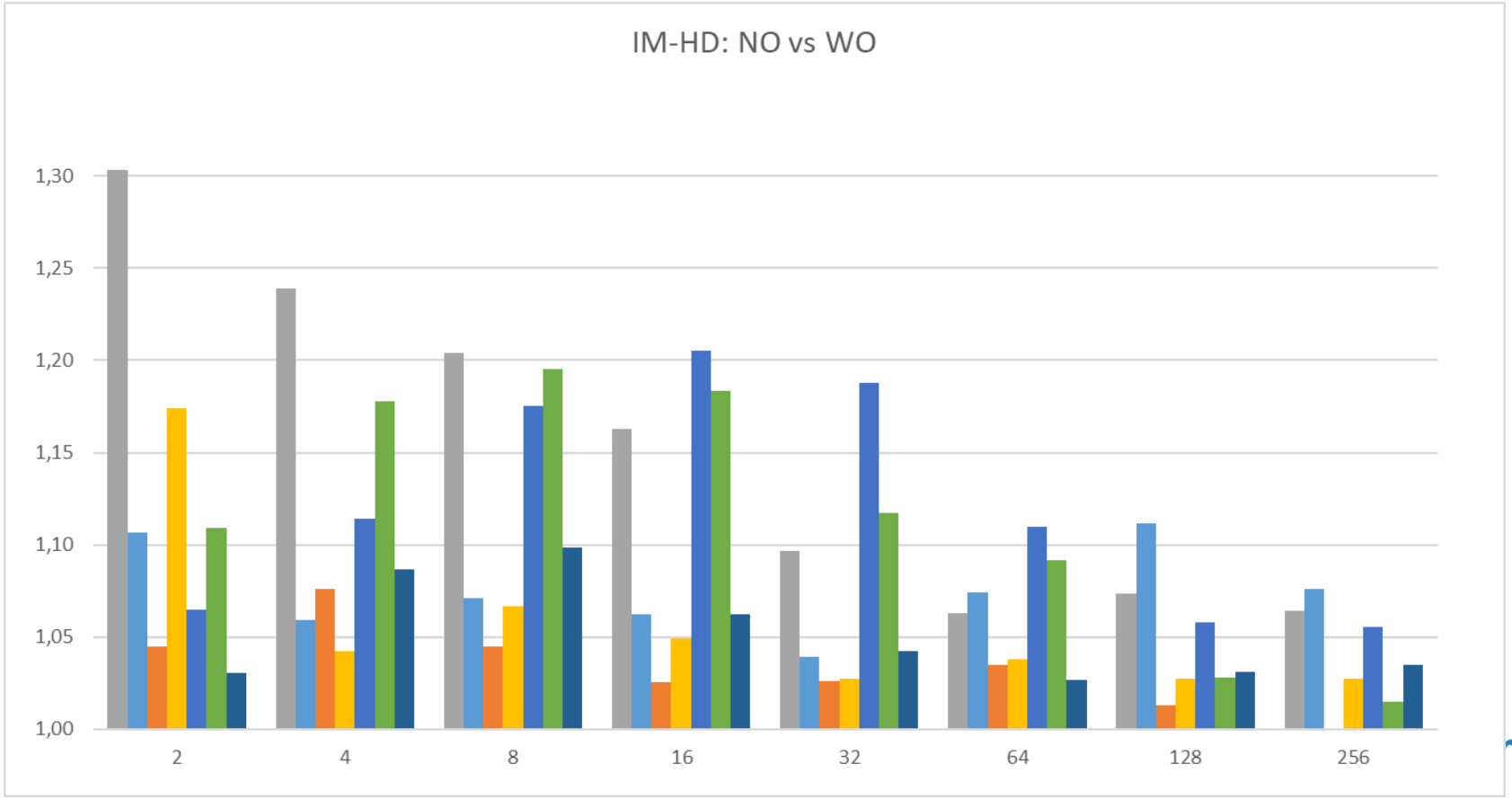
Another way to look at the numbers?

```
> SELECT [...];
+-----------+------------+
| thread_id | COUNT_STAR |
+-----------+------------+
|     63581 |        394 |
|     63582 |        292 |
|     63583 |        272 |
|     63584 |        260 |
|     63585 |        251 |
|     63586 |        237 |
[...]
|     63606 |        108 |
|     63607 |        106 |
|     63608 |        104 |
|     63609 |        100 |
|     63610 |         98 |
|     63611 |         94 |
|     63612 |         89 |
+-----------+------------+
32 rows in set (0.00 sec)
```

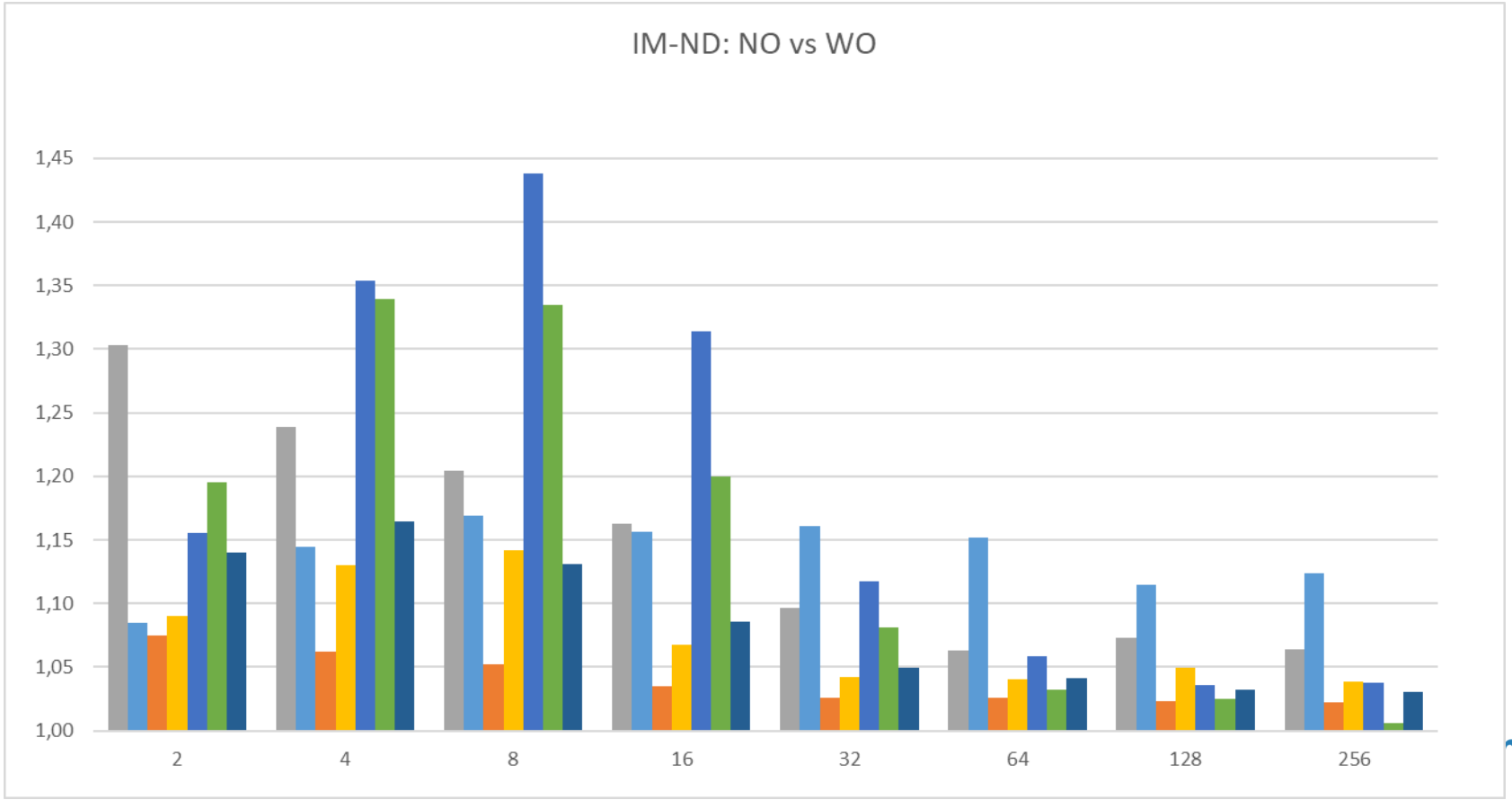
```
> SELECT [...];
+-----------+------------+
| thread_id | COUNT_STAR |
+-----------+------------+
|     63627 |        749 |
|     63628 |        504 |
|     63629 |        457 |
|     63630 |        433 |
|     63631 |        405 |
|     63632 |        384 |
[...]
|     63684 |         73 |
|     63685 |         73 |
|     63686 |         71 |
|     63687 |         68 |
|     63688 |         65 |
|     63689 |         65 |
|     63690 |         62 |
+-----------+------------+
64 rows in set (0.00 sec)
```

```
> SELECT [...];
+-----------+------------+
| thread_id | COUNT_STAR |
+-----------+------------+
|     63704 |        711 |
|     63705 |        468 |
|     63706 |        420 |
|     63707 |        378 |
|     63708 |        359 |
|     63709 |        340 |
[...]
|     63825 |          3 |
|     63826 |          3 |
|     63827 |          3 |
|     63828 |          3 |
|     63829 |          3 |
|     63830 |          2 |
|     63831 |          2 |
+-----------+------------+
128 rows in set (0.00 sec)
```

MySQL 8.0 – What about commit order ?



MySQL 8.0 – What about commit order ?’



MySQL 5.7 and 8.0 // Repl. Summary

- Parallel replication in MySQL 5.7 is not simple:
  - Need precise tuning
  - Long transactions block the parallel replication pipeline
  - Care about Intermediate masters
- Write Set in MySQL 8.0 gives very interesting results:
  - No problem with Intermediate masters
  - Allows to test with Intermediate Master
  - Some great speedups and most of them very good

And please
test by yourself
and share results

// Replication: Links

- Replication crash safety with MTS in MySQL 5.6 and 5.7: reality or illusion?
  https://jfg-mysql.blogspot.com/2016/01/replication-crash-safety-with-mts.html
- A Metric for Tuning Parallel Replication in MySQL 5.7
  https://jfg-mysql.blogspot.com/2017/02/metric-for-tuning-parallel-replication-mysql-5-7.html

- Solving MySQL Replication Lag with LOGICAL_CLOCK and Calibrated Delay
  https://www.vividcortex.com/blog/solving-mysql-replication-lag-with-logical_clock-and-calibrated-delay
- How to Fix a Lagging MySQL Replication
  https://thoughts.t37.net/fixing-a-very-lagging-mysql-replication-db6eb5a6e15d

- Binlog Servers:
  - http://blog.booking.com/mysql_slave_scaling_and_more.html
  - Better Parallel Replication for MySQL: http://blog.booking.com/better_parallel_replication_for_mysql.html
  - http://blog.booking.com/abstracting_binlog_servers_and_mysql_master_promotion_wo_reconfiguring_slaves.html

// Replication: Links’

- An update on Write Set (parallel replication) bug fix in MySQL 8.0
  https://jfg-mysql.blogspot.com/2018/01/an-update-on-write-set-parallel-replication-bug-fix-in-mysql-8-0.html
- Write Set in MySQL 5.7: Group Replication
  https://jfg-mysql.blogspot.com/2018/01/write-set-in-mysql-5-7-group-replication.html
- More Write Set in MySQL: Group Replication Certification
  https://jfg-mysql.blogspot.com/2018/01/more-write-set-in-mysql-5-7-group-replication-certification.html

// Replication: Links”

- Bugs/feature requests:
  - The doc. of slave-parallel-type=LOGICAL_CLOCK wrongly reference Group Commit: Bug#85977
  - Allow slave_preserve_commit_order without log-slave-updates: Bug#75396
  - MTS with slave_preserve_commit_order not repl. crash safe: Bug#80103
  - Automatic Repl. Recovery Does Not Handle Lost Relay Log Events: Bug#81840
  - Expose, on the master/slave, counters for monitoring // info. quality: Bug#85965 & Bug#85966
  - Expose counters for monitoring Write Set barriers: Bug#86060
  - Deadlock with slave_preserve_commit_order=ON with Bug#86078: Bug#86079 & Bug#89247
- Fixed bugs:
  - Message after MTS crash misleading: Bug#80102 (and Bug#77496)
  - Replication position lost after crash on MTS configured slave: Bug#77496
  - Full table scan bug in InnoDB: MDEV-10649, Bug#82968 and Bug#82969
  - The function reset_connection does not reset Write Set in WRITESET_SESSION: Bug#86063
  - Bad Write Set tracking with UNIQUE KEY on a DELETE followed by an INSERT: Bug#86078

Thanks

Eduardo Ortega (MySQL Database Engineer)
eduardo DOT ortega AT booking.com

Jean-François Gagné (System Engineer)
jeanfrancois DOT gagne AT booking.com