# 20 Tips for MySQL Performance Tuning

## Unlocking the True Potential of Your Database

*Frédéric Descamps*
*Community Manager*
*Oracle MySQL*
*MySQL 30Y Virtual Event - 2025*

# Who am I ?

*about.me/lefred*

# Frédéric Descamps

- 𝕏 *@lefred*
- *@lefredbe.bsky.social*
- *@lefred@fosstodon.org*
- *MySQL Evangelist*
- *using MySQL since version 3.20*
- *devops believer*
- *living in* 🇧🇪
- *https://lefred.be*

# Disclaimer

*We will use a fake test database called* `ecommerce` *for our examples.*

*The design of the tables and the queries are not optimized on purpose.*

```
+----------------------+
| Tables_in_ecommerce  |
+----------------------+
| invoice_status       |
| order_items          |
| orders               |
| products             |
| reviews              |
| users                |
+----------------------+
```

# Know your data

*engines, size*

# Don't use MyISAM !


*Do not ever use* `CREATE TABLE... ENGINE=MyISAM` *anymore !*

*Please!! Please!! Please!!*

```
SQL> set persistdefault_storage_engine="InnoDB";
SQL> set persist_only disabled_storage_engines="MyISAM";
```

*In MySQL 8.x, defaults are strict and InnoDB guarantees DURABILITY !*

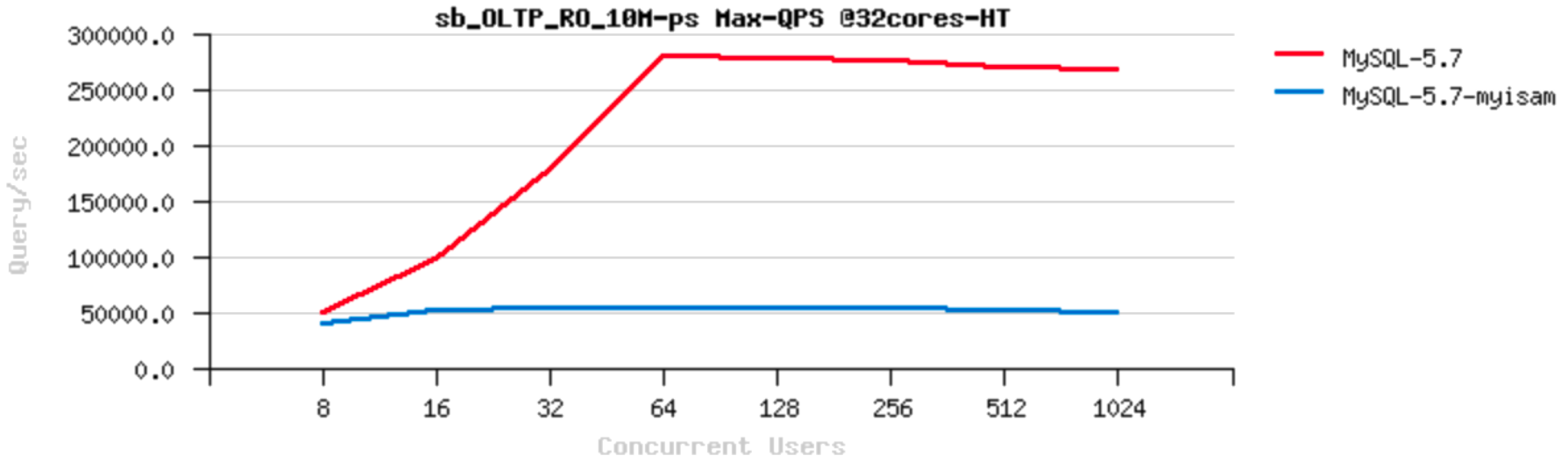
*Keep your data safe !*

# Always use InnoDB !!

- *ACID*
- *multiple lock types*
- *faster than MyISAM*

**Mixed OLTP_RO workload**

dataset 10M x 1-table @32cores-HT :



# Know your data - storage engines

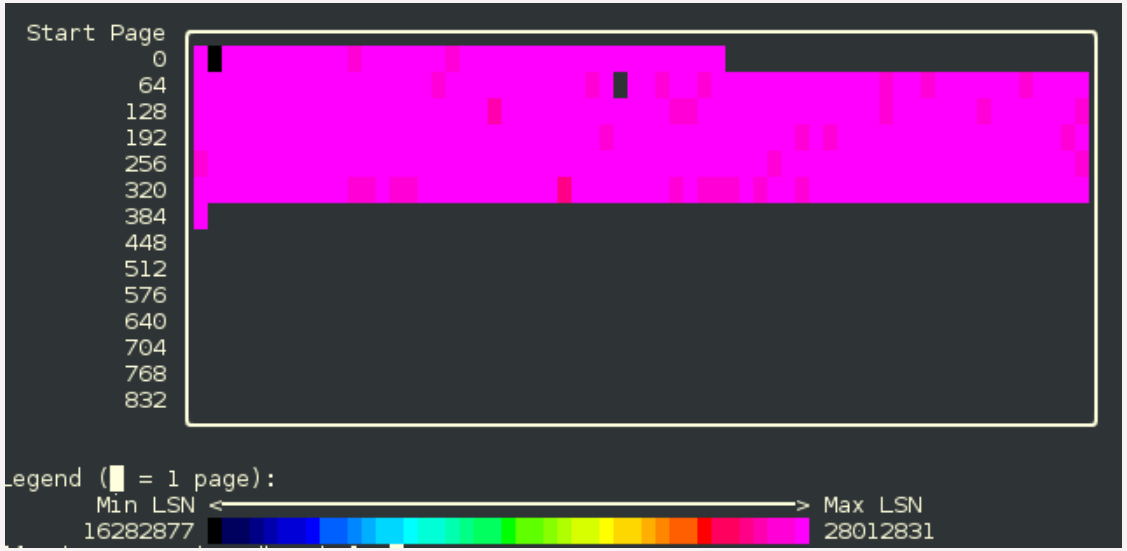
*Check the storage engines and the size of your tables:*

```
SQL> select concat(table_schema, '.', table_name) as 'TABLE', ENGINE,
            format(table_rows,0) `ROWS`, format_bytes(data_length) DATA,
            format_bytes(index_length) IDX, format_bytes(data_length + index_length) 'TOTAL SIZE'
      from information_schema.tables  where table_schema='ecommerce'
      order by data_length + index_length;
+--------------------------+--------+--------+------------+------------+------------+
| TABLE                    | ENGINE | ROWS   | DATA       | IDX        | TOTAL SIZE |
+--------------------------+--------+--------+------------+------------+------------+
| ecommerce.invoice_status | MyISAM | 1,557  | 30.41 KiB  | 1.00 KiB   | 31.41 KiB  |
| ecommerce.orders         | InnoDB | 1,259  | 112.00 KiB | 80.00 KiB  | 192.00 KiB |
| ecommerce.order_items    | InnoDB | 6,289  | 288.00 KiB | 256.00 KiB | 544.00 KiB |
| ecommerce.reviews        | InnoDB | 8,955  | 2.52 MiB   | 800.00 KiB | 3.30 MiB   |
| ecommerce.users          | InnoDB | 3,901  | 1.52 MiB   | 2.34 MiB   | 3.86 MiB   |
| ecommerce.products       | InnoDB | 39,744 | 5.52 MiB   |    0 bytes | 5.52 MiB   |
+--------------------------+--------+--------+------------+------------+------------+
```

# Know your data - primary keys

*Bad primary keys can slow down your queries and your writes. It also impacts your secondary indexes and can massively grow your data size and IOPS on disk.*

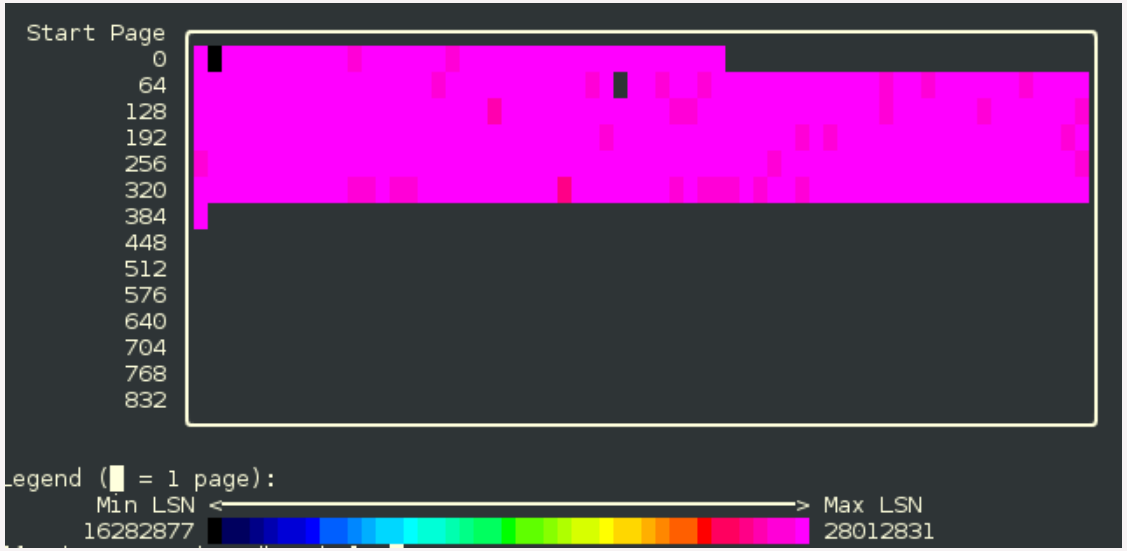
*Not having primary keys is even worse and can cause contention problems (`dict_sys->mutex`) and replication lag.*

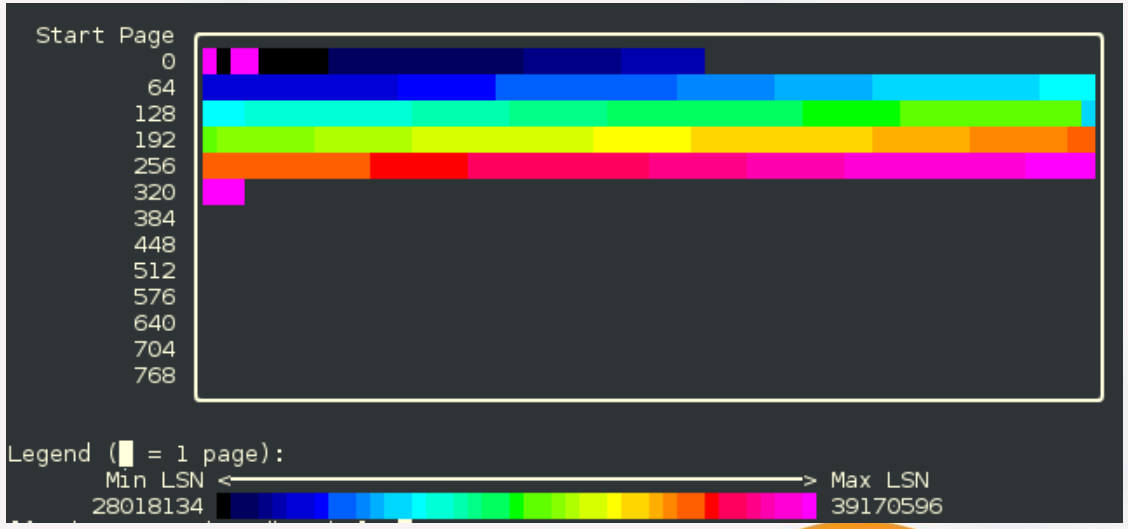


# Know your data - primary keys

*Bad primary keys can slow down your queries and your writes. It also impacts your secondary indexes and can massively grow your data size and IOPS on disk.*

*Not having primary keys is even worse and can cause contention problems (`dict_sys->mutex`) and replication lag.*



# Know your data - primary keys (2)

*Find tables without primary keys:*

```
SQL> select i.table_id, t.name
     from information_schema.innodb_indexes i
     join information_schema.innodb_tables t on (i.table_id = t.table_id)
     where i.name='GEN_CLUST_INDEX' and t.name like 'ecommerce/%';
+----------+--------------------+
| table_id | name               |
+----------+--------------------+
|     3049 | ecommerce/reviews |
+----------+--------------------+
1 row in set (0.0053 sec)
```

# Know your data - primary keys (3)

*Let's verify the definition of this table:*

```
SQL> show create table reviews\G
 *************************** 1. row ***************************
       Table: reviews
Create Table: CREATE TABLE `reviews` (
  `user_id` varchar(36) DEFAULT NULL,
  `product_id` int DEFAULT NULL,
  `rating` int DEFAULT NULL,
  `review` text,
  `created_at` datetime DEFAULT CURRENT_TIMESTAMP,
  KEY `user_id` (`user_id`),
  KEY `product_id` (`product_id`),
  CONSTRAINT `reviews_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users` (`id`),
  CONSTRAINT `reviews_ibfk_2` FOREIGN KEY (`product_id`) REFERENCES `products` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.0024 sec)
```

# Know your data - primary keys (4)

*Let's create a new identical table but first we will enable the generation of an invisible primary key when none is provided:*

```
SQL> set persist sql_generate_invisible_primary_key=1;
Query OK, 0 rows affected (0.0005 sec)

SQL> CREATE TABLE `reviews2` (   `user_id` varchar(36) DEFAULT NULL,
`product_id` int DEFAULT NULL,   `rating` int DEFAULT NULL,   `review` text,
`created_at` datetime DEFAULT CURRENT_TIMESTAMP,   KEY `user_id` (`user_id`),
KEY `product_id` (`product_id`),   CONSTRAINT `reviews2_ibfk_1`
FOREIGN KEY (`user_id`) REFERENCES `users` (`id`),   CONSTRAINT `reviews2_ibfk_2`
FOREIGN KEY (`product_id`) REFERENCES `products` (`id`) ) ENGINE=InnoDB;
Query OK, 0 rows affected (0.0617 sec)
```

# Know your data - primary keys (5)

*Let's verify:*

```
SQL> desc reviews2;
+------------+-----------------+------+-----+-------------------+----------------------------+
| Field      | Type            | Null | Key | Default           | Extra                      |
+------------+-----------------+------+-----+-------------------+----------------------------+
| my_row_id  | bigint unsigned | NO   | PRI | NULL              | auto_increment INVISIBLE   |
| user_id    | varchar(36)     | YES  | MUL | NULL              |                            |
| product_id | int             | YES  | MUL | NULL              |                            |
| rating     | int             | YES  |     | NULL              |                            |
| review     | text            | YES  |     | NULL              |                            |
| created_at | datetime        | YES  |     | CURRENT_TIMESTAMP | DEFAULT_GENERATED          |
+------------+-----------------+------+-----+-------------------+----------------------------+
6 rows in set (0.0018 sec)
```

# Know your data - primary keys (6)

*And we can now copy the data and swap the tables:*

```
SQL> insert into reviews2 select * from reviews;
Query OK, 8991 rows affected (0.4290 sec)

SQL> drop table reviews;
Query OK, 0 rows affected (0.0269 sec)

SQL> rename table reviews2 to reviews;
Query OK, 0 rows affected (0.0311 sec)
```

# Know your data - primary keys (7)

*Let's query the table:*

```
SQL> select r.* from reviews r limit 1\G
 *************************** 1. row ***************************
   user_id: b5f2bb5b-f9a1-11ef-8422-5e8693515ddb
product_id: 6735
    rating: 4
    review: Somebody mention deep Republican animal sister ...
created_at: 2025-03-05 10:22:37
```

# Know your data - primary keys (8)

*And this time if we specify* `my_row_id` *in the query:*

```
SQL> select my_row_id, r.* from reviews r limit 1\G
 *************************** 1. row ***************************
  my_row_id: 1
    user_id: b5f2bb5b-f9a1-11ef-8422-5e8693515ddb
 product_id: 6735
     rating: 4
     review: Somebody mention deep Republican animal sister ...
 created_at: 2025-03-05 10:22:37
```

# Know your data - primary keys (3)

*Find tables with an eventual bad primary key:*

```
SQL> select a.TABLE_SCHEMA,a.TABLE_NAME, b.ENGINE, a.COLUMN_NAME, a.DATA_TYPE,
            a.COLUMN_TYPE, a.COLUMN_KEY, b.TABLE_ROWS
     from information_schema.COLUMNS as a
     join information_schema.TABLES as b using (table_name,table_schema)
     where COLUMN_KEY='PRI' and ENGINE="InnoDB" and DATA_TYPE not like '%int'
       and DATA_TYPE not like 'enum%'   and DATA_TYPE not like 'date%'
       and DATA_TYPE not like 'time%'   and a.table_schema='ecommerce';
+--------------+------------+--------+-------------+-----------+-------------+------------+------------+
| TABLE_SCHEMA | TABLE_NAME | ENGINE | COLUMN_NAME | DATA_TYPE | COLUMN_TYPE | COLUMN_KEY | TABLE_ROWS |
+--------------+------------+--------+-------------+-----------+-------------+------------+------------+
| ecommerce    | users      | InnoDB | id          | varchar   | varchar(36) | PRI        |       3901 |
+--------------+------------+--------+-------------+-----------+-------------+------------+------------+
1 row in set (0.0012 sec)
```

# Know your data - primary keys (4)

```
SQL> show create table ecommerce.users\G
 *************************** 1. row ***************************
       Table: users
Create Table: CREATE TABLE `users` (
  `id` varchar(36) NOT NULL,
  `name` text,
  `email` text,
  `address` varchar(200) DEFAULT NULL,
  `city` varchar(100) DEFAULT NULL,
  `country` varchar(100) DEFAULT NULL,
  `created_at` datetime DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  KEY `name_idx` (`name`(100)),
  KEY `address_idx` (`address`),
  KEY `city_idx` (`city`),
  KEY `country_idx` (`country`),
  KEY `city_country_idx` (`city`,`country`),
  KEY `name_email_idx` (`name`(100),`email`(100))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.0024 sec)
```

# Know your data - primary keys (5)

```
SQL> select id, name, created_at from users limit 5;
+--------------------------------------+-----------------+---------------------+
| id                                   | name            | created_at          |
+--------------------------------------+-----------------+---------------------+
| 02ecfa36-f3b0-11ef-9704-5e1b9081e705 | Katie Travis    | 2025-02-25 20:37:56 |
| 02ed1c92-f3b0-11ef-9704-5e1b9081e705 | James Henderson | 2025-02-25 20:37:56 |
| 02ed383e-f3b0-11ef-9704-5e1b9081e705 | William Mercer  | 2025-02-25 20:37:56 |
| 02ed59b2-f3b0-11ef-9704-5e1b9081e705 | Carla Manning   | 2025-02-25 20:37:56 |
| 02ed721d-f3b0-11ef-9704-5e1b9081e705 | Benjamin Ashley | 2025-02-25 20:37:56 |
+--------------------------------------+-----------------+---------------------+
5 rows in set (0.0008 sec)
```

# Know your data - primary keys (6)

```
SQL> select id, name, created_at from users order by created_at limit 5;
+--------------------------------------+-------------------+---------------------+
| id                                   | name              | created_at          |
+--------------------------------------+-------------------+---------------------+
| 8a028fc4-f35a-11ef-9704-5e1b9081e705 | Kimberly Atkinson | 2025-02-25 10:26:05 |
| 8a02cb4f-f35a-11ef-9704-5e1b9081e705 | Betty Davis       | 2025-02-25 10:26:05 |
| 8a02ac38-f35a-11ef-9704-5e1b9081e705 | David Rios        | 2025-02-25 10:26:05 |
| 8a02e292-f35a-11ef-9704-5e1b9081e705 | Brooke Weber      | 2025-02-25 10:26:05 |
| 8a0279ce-f35a-11ef-9704-5e1b9081e705 | Carlos Harris     | 2025-02-25 10:26:05 |
+--------------------------------------+-------------------+---------------------+
5 rows in set (0.0038 sec)
```

# Know your data - primary keys (7)

*Store UUIDs as BINARY(16) and swap the timestamp to have incremental values:*

```
SQL> select uuid_to_bin('02ecfa36-f3b0-11ef-9704-5e1b9081e705', 1);
+--------------------------------------------------------+
| uuid_to_bin('02ecfa36-f3b0-11ef-9704-5e1b9081e705', 1) |
+--------------------------------------------------------+
| 0x11EFF3B002ECFA3697045E1B9081E705                     |
+--------------------------------------------------------+

SQL> select uuid_to_bin('8a028fc4-f35a-11ef-9704-5e1b9081e705', 1);
+--------------------------------------------------------+
| uuid_to_bin('8a028fc4-f35a-11ef-9704-5e1b9081e705', 1) |
+--------------------------------------------------------+
| 0x11EFF35A8A028FC497045E1B9081E705                     |
+--------------------------------------------------------+
```

# Know your data - use the right data type

*Use the right data type for your columns, and don't over size them.*

*Don't store numbers in* `VARCHAR`*, use* `INT`, `BIGINT`, `DECIMAL` *or* `FLOAT`*.*

*Use* `ENUM` *for columns with a limited number of values.*

*Use* `JSON` *for columns with a variable number of attributes and don't use* `TEXT` *to store your JSON.*

*Try to avoid* `TEXT`, `BLOB`, `VARCHAR(255)` *and use the smallest data type possible.*

# Know your data - use the right data type (2)

```
SQL> select table_name, column_name, data_type, column_type, character_maximum_length
     from information_schema.columns where TABLE_SCHEMA='ecommerce'
      and data_type in ('blob', 'text', 'varchar') order by 5 desc;
+----------------+-------------+-----------+--------------+--------------------------+
| TABLE_NAME     | COLUMN_NAME | DATA_TYPE | COLUMN_TYPE  | CHARACTER_MAXIMUM_LENGTH |
+----------------+-------------+-----------+--------------+--------------------------+
| products       | description | text      | text         |                    65535 |
| reviews        | review      | text      | text         |                    65535 |
| users          | name        | text      | text         |                    65535 |
| users          | email       | text      | text         |                    65535 |
| products       | name        | varchar   | varchar(255) |                      255 |
| users          | address     | varchar   | varchar(200) |                      200 |
| users          | city        | varchar   | varchar(100) |                      100 |
| users          | country     | varchar   | varchar(100) |                      100 |
| orders         | user_id     | varchar   | varchar(36)  |                       36 |
| reviews        | user_id     | varchar   | varchar(36)  |                       36 |
| users          | id          | varchar   | varchar(36)  |                       36 |
| invoice_status | status      | varchar   | varchar(10)  |                       10 |
+----------------+-------------+-----------+--------------+--------------------------+
12 rows in set (0.0009 sec)
```

# Know your data - use the right data type (3)

*Let's have a look at our first case (`products.description`):*

```
SQL> select CHAR_LENGTH(description) from products order by 1 desc limit 5;
+---------------------------+
| CHAR_LENGTH(description) |
+---------------------------+
|                       118 |
|                       118 |
|                       117 |
|                       116 |
|                       115 |
+---------------------------+
5 rows in set (0.0312 sec)
```

# Know your data - use the right data type (3)

*Let's have a look at our first case (`products.description`):*

```
SQL> select CHAR_LENGTH(description) from products order by 1 desc limit 5;
+--------------------------+
| CHAR_LENGTH(description) |
+--------------------------+
|                      118 |
```

*Let's fix this:*

```
SQL> alter table products modify description varchar(150);
```

```
5 rows in set (0.0312 sec)
```

# Know your data - use the right data type (4)

*And what about* `invoice_status.status` *?*

```
SQL> select status, count(*) from invoice_status group by status;
+--------+----------+
| status | count(*) |
+--------+----------+
| sent   |     1258 |
| paid   |      299 |
+--------+----------+
2 rows in set (0.0015 sec)
```

# Know your data - use the right data type (4)

*And what about* `invoice_status.status` *?*

```
SQL> select status, count(*) from invoice_status group by status;
+--------+----------+
| status | count(*) |
+--------+----------+
| sent   |     1258 |
```

*We could use an* `ENUM`*:*

```
SQL> alter table invoice_status modify status ENUM('sent', 'paid');
```

# Recap: don't use MyISAM - TIP 1

```
SQL> set persist default_storage_engine="InnoDB";
SQL> set persist_only disabled_storage_engines="MyISAM";

SQL> alter table <table_name> engine=InnoDB;
```

# Recap: use Primary Keys, always! - TIP 2

```
SQL> set persist sql_require_primary_key=1;
SQL> set persist sql_generate_invisible_primary_key=1;
```

*And keep your primary keys as small as possible, a good choice is:*

`BIGINT UNSIGNED NOT NULL AUTO_INCREMENT`

# Recap: use Primary Keys, always! - TIP 2

```
SQL> set persist sql_require_primary_key=1;
SQL> set persist sql_generate_invisible_primary_key=1;
```

*And keep your primary keys as small as possible, a good choice is:*

`BIGINT UNSIGNED NOT NULL AUTO_INCREMENT`

```
SQL> select table_name, column_name, data_type, column_type, extra
      from information_schema.columns
     where TABLE_SCHEMA='ecommerce' and data_type ='int'
       and column_type not like '%unsigned' and column_key='PRI';
```

# Recap: store UUIDs as BINARY(16) and swap the timestamp to have incremental values - TIP 3

```
uuid BINARY(16) DEFAULT (UUID_TO_BIN(UUID(), 1)) PRIMARY KEY
```

# Recap: use the right data type and keep them as small as possible - TIP 4

*Don't create such table:*

```
CREATE TABLE `products` (
  `id` VARCHAR(255) NOT NULL,
  `name` TEXT DEFAULT NULL,
  `description` TEXT DEFAULT NULL,
  `price` TEXT NOT NULL,
  `created_at` datetime DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB;
```

# Indexes are important

*not too much and not too few*

# Check your indexes

*It's important to not maintain unused indexes, this can slow down write operations and load the Optimizer for the QEP creation.*

# Check your indexes

*It's important to not maintain unused indexes, this can slow down write operations and load the Optimizer for the QEP creation.*

*And it's the same for duplicate indexes !*

# Check your indexes

*It's important to not maintain unused indexes, this can slow down write operations and load the Optimizer for the QEP creation.*

*And it's the same for duplicate indexes !*

*And finally, you may also miss some indexes causing full tables scans :-(*

# Check your indexes

*It's important to not maintain unused indexes, this can slow down write operations and load the Optimizer for the QEP creation.*

*And it's the same for duplicate indexes !*

*And finally, you may also miss some indexes causing full tables scans :-(*

*MySQL provides you useful information through* `sys` *schema.*

# Unused Indexes

```
SQL> select database_name, table_name, t1.index_name,
       format_bytes(stat_value * @@innodb_page_size) size
  from sys.schema_unused_indexes t2
  join mysql.innodb_index_stats t1 on object_schema=database_name
   and object_name=table_name and t2.index_name=t1.index_name
 where stat_name='size' and database_name = 'ecommerce' order by stat_value desc;
```

# Unused Indexes

```
SQL> select database_name, table_name, t1.index_name,
       format_bytes(stat_value * @@innodb_page_size) size
  from sys.schema_unused_indexes t2
  join mysql.innodb_index_stats t1 on object_schema=database_name
   and object_name=table_name and t2.index_name=t1.index_name
 where stat_name='size' and database_name = 'ecommerce' order by stat_value desc;
```

```
+---------------+-------------+------------------+------------+
| database_name | table_name  | index_name       | size       |
+---------------+-------------+------------------+------------+
| ecommerce     | users       | address_idx      | 1.52 MiB   |
| ecommerce     | users       | city_idx         | 1.52 MiB   |
| ecommerce     | users       | country_idx      | 1.52 MiB   |
| ecommerce     | users       | city_country_idx | 1.52 MiB   |
| ecommerce     | users       | name_email_idx   | 1.52 MiB   |
| ecommerce     | reviews     | user_id          | 1.52 MiB   |
| ecommerce     | order_items | product_id       | 256.00 KiB |
+---------------+-------------+------------------+------------+
```

# Duplicate Indexes

```
SQL> select t2.*, format_bytes(stat_value * @@innodb_page_size) size
  from mysql.innodb_index_stats t1
   join sys.schema_redundant_indexes t2
    on table_schema=database_name and t2.table_name=t1.table_name
   and t2.redundant_index_name=t1.index_name
 where stat_name='size' and database_name = 'ecommerce' order by stat_value desc\G
```

```
*************************** 1. row ***************************
              table_schema: ecommerce
                table_name: users
      redundant_index_name: city_idx
   redundant_index_columns: city
redundant_index_non_unique: 1
       dominant_index_name: city_country_idx
    dominant_index_columns: city,country
 dominant_index_non_unique: 1
           subpart_exists: 0
           sql_drop_index: ALTER TABLE `ecommerce`.`users` DROP INDEX `city_idx`
                     size: 1.52 MiB
*************************** 2. row ***************************
              table_schema: ecommerce
                table_name: users
      redundant_index_name: name_idx
   redundant_index_columns: name
redundant_index_non_unique: 1
       dominant_index_name: name_email_idx
    dominant_index_columns: name,email
 dominant_index_non_unique: 1
           subpart_exists: 1
           sql_drop_index: ALTER TABLE `ecommerce`.`users` DROP INDEX `name_idx`
                     size: 528.00 KiB
```

# Best Practices to drop indexes

*Instead of dropping an index, it's recommended to set it as invisible for a while and check if it's really not used.*

*When this is confirmed, you can drop it.*

```
SQL> alter table users alter index city_idx invisible;
```

# Best Practices to drop indexes (2)

*To list all invisible indexes:*

```
SQL> select TABLE_NAME, INDEX_NAME, IS_VISIBLE from INFORMATION_SCHEMA.STATISTICS
     where TABLE_SCHEMA = 'ecommerce' and IS_VISIBLE='no';
+------------+------------+------------+
| TABLE_NAME | INDEX_NAME | IS_VISIBLE |
+------------+------------+------------+
| users      | city_idx   | NO         |
+------------+------------+------------+
1 row in set (0.0013 sec)
```

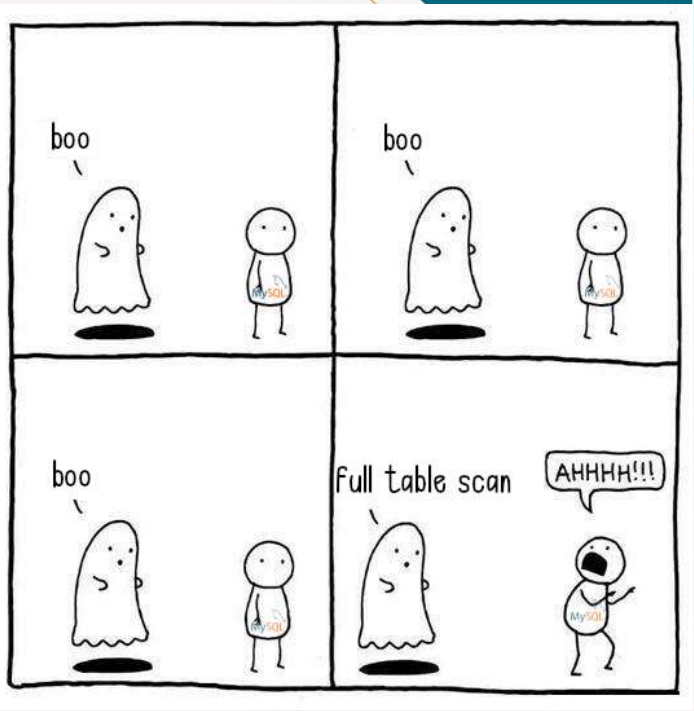
# Best Practices to drop indexes (3)

*You can also use* `SHOW INDEX FROM` *if you prefer:*

```
SQL> show index from  users where visible='no'\G
 *************************** 1. row ***************************
        Table: users
   Non_unique: 1
     Key_name: city_idx
 Seq_in_index: 1
  Column_name: city
    Collation: A
  Cardinality: 963
     Sub_part: NULL
       Packed: NULL
         Null: YES
   Index_type: BTREE
      Comment:
Index_comment:
      Visible: NO
   Expression: NULL
```

# Missing Indexes

```
SQL> select * from sys.schema_tables_with_full_table_scans
    where object_schema='ecommerce';
+---------------+-------------+-------------------+----------+
| object_schema | object_name | rows_full_scanned | latency  |
+---------------+-------------+-------------------+----------+
| ecommerce     | products    |         135320995 | 1.13 min |
| ecommerce     | users       |          19139037 | 10.25 s  |
| ecommerce     | order_items |          10075569 | 2.71 s   |
| ecommerce     | orders      |           6100757 | 1.81 s   |
+---------------+-------------+-------------------+----------+
```



# Missing Indexes (2)

```
SQL > select t1.*, query_sample_text from sys.statements_with_full_table_scans t1
      join performance_schema.events_statements_summary_by_digest using(digest)
      where query like '%products%'\G
 *************************** 1. row ***************************
                 query: SELECT * FROM `ecommerce` . `products` WHERE `price` > ?
                    db: NULL
            exec_count: 4816
         total_latency: 1.19 min
     no_index_used_count: 4816
no_good_index_used_count: 0
       no_index_used_pct: 100
               rows_sent: 90978218
           rows_examined: 91909000
           rows_sent_avg: 18891
       rows_examined_avg: 19084
              first_seen: 2025-02-24 20:20:17.480185
               last_seen: 2025-02-25 22:01:18.556123
                  digest: c330311f85f465c26e40ae506313b8a02589fd3c8e59742c30dc0179e25cbd16
       query_sample_text: SELECT * FROM ecommerce.products WHERE price>100
```

# Best Practices to add indexes

*The amount of parallel threads used by InnoDB is controlled by* `innodb_ddl_threads`.

*This new variable is coupled with another new variable:* `innodb_ddl_buffer_size`.

*If you have fast storage and multiple CPU cores, tuning these variables can speed up secondary index creation.*

# Parallel Index Creation - example

```
SQL > alter table booking
      add index idx_2(flight_id, seat, passenger_id);
Query OK, 0 rows affected (9 min 0.6838 sec)
```

# Parallel Index Creation - example

```
SQL > alter table booking
      add index idx_2(flight_id, seat, passenger_id);
Query OK, 0 rows affected (9 min 0.6838 sec)
```

*The default settings are:*

```
innodb_ddl_threads = 4
innodb_ddl_buffer_size = 1048576
innodb_parallel_read_threads = 4
```

# Parallel Index Creation - example

```
SQL > alter table booking
      add index idx_2(flight_id, seat, passenger_id);
Query OK, 0 rows affected (9 min 0.6838 sec)
```

*The default settings are:*

```
innodb_ddl_threads = 4
innodb_ddl_buffer_size = 1048576
innodb_parallel_read_threads = 4
```

*The* **`innodb_ddl_buffer_size`** *is shared between all* **`innodb_ddl_threads`** *defined. If you increase the amount of threads, I recommend that you also increase the buffer size.*

# Parallel Index Creation - example (2)

*To find the best values for these variables, let's have a look at the amount of CPU cores:*

```
SQL > select count from information_schema.INNODB_METRICS
      where name = 'cpu_n';
+-------+
| count |
+-------+
|    16 |
+-------+
```

*We have then 16 cores to share. As my machine as plenty of memory, I will allocate 1GB for the InnoDB DDL buffer.*

# Parallel Index Creation - example (3)

```
SQL> set innodb_ddl_threads = 8;
SQL> set innodb_parallel_read_threads = 8;
SQL> set innodb_ddl_buffer_size = 1048576000;
```

# Parallel Index Creation - example (3)

```
SQL> set innodb_ddl_threads = 8;
SQL> set innodb_parallel_read_threads = 8;
SQL> set innodb_ddl_buffer_size = 1048576000;
```

*We can now retry the same index creation as previously:*

```
SQL>  alter table booking add index idx_2(flight_id, seat, passenger_id);
Query OK, 0 rows affected (3 min 9.1862 sec)
```

# Parallel Index Creation - example (4)

*I recommend to make tests to define the optimal settings for your database, your hardware and data.*

*For example, on my system, I got the best result setting the buffer size to 2GB and both ddl threads and parallel read threads to 4.*

*It took 2 min 43 sec, much better than the initial 9 minutes !*

# Recap: don't maintain useless indexes - TIP 5

- *remove duplicate indexes*
- *remove unused indexes (set them invisible first)*

# Recap: add missing indexes and do it fast! - TIP 6

- *check for full table scans*
- *tune the parallel index creation*

```
SQL> set innodb_ddl_threads = 8;
SQL> set innodb_parallel_read_threads = 8;
SQL> set innodb_ddl_buffer_size = 1048576000;
```

# Workload

*what are my queries?*

# What is my workload?

*Many people don't really know if their workload is read or write intensive, or they think they know!*

*Let's see how to find out:*

```
SQL> select sum(count_read) `tot reads`,
            concat(round((sum(count_read)/sum(count_star))*100, 2),"%") `reads`,
            sum(count_write) `tot writes`,
            concat(round((sum(count_write)/sum(count_star))*100, 2),"%") `writes`
     from performance_schema.table_io_waits_summary_by_table
     where count_star > 0 and object_schema='ecommerce' ;
+------------+---------+------------+--------+
| tot reads  | reads   | tot writes | writes |
+------------+---------+------------+--------+
| 1334124400 | 100.00% |      51692 | 0.00%  |
+------------+---------+------------+--------+
```

# What is my workload? (2)

*And we can check by tables:*

```
SQL> select object_schema, object_name,
            concat(round((count_read/count_star)*100, 2),"%") `reads`,
            concat(round((count_write/count_star)*100, 2),"%") `writes`
     from performance_schema.table_io_waits_summary_by_table
     where count_star > 0 and object_schema='ecommerce' ;
+---------------+----------------+---------+---------+
| object_schema | object_name    | reads   | writes  |
+---------------+----------------+---------+---------+
| ecommerce     | users          | 99.99%  | 0.01%   |
| ecommerce     | orders         | 99.96%  | 0.04%   |
| ecommerce     | order_items    | 99.89%  | 0.11%   |
| ecommerce     | reviews        | 99.97%  | 0.03%   |
| ecommerce     | products       | 100.00% | 0.00%   |
| ecommerce     | invoice_status | 0.00%   | 100.00% |
+---------------+----------------+---------+---------+
```

# Find the Ugly Duckling

*If you should optimize only one query, the best candidate should be the query that consumes the most of the execution time (seen as latency in PFS, but usually called "response time").*

# Find the Ugly Duckling

*If you should optimize only one query, the best candidate should be the query that consumes the most of the execution time (seen as latency in PFS, but usually called "response time").*

`sys Schema` *contains all the necessary info to find that Ugly Duckling:*

# Find the Ugly Duckling

*If you should optimize only one query, the best candidate should be the query that consumes the most of the execution time (seen as latency in PFS, but usually called "response time").*

`sys Schema` *contains all the necessary info to find that Ugly Duckling:*

```
SELECT schema_name, format_pico_time(total_latency) tot_lat,
       exec_count, format_pico_time(total_latency/exec_count) latency_per_call,
       query_sample_text
  FROM sys.x$statements_with_runtimes_in_95th_percentile AS t1
  JOIN performance_schema.events_statements_summary_by_digest AS t2
    ON t2.digest=t1.digest
 WHERE schema_name = 'ecommerce'
ORDER BY (total_latency/exec_count) desc LIMIT 1\G
```

# Find the Ugly Duckling

*If you should optimize only one query, the best candidate should be the query that consumes the most of the execution time (seen as latency in PFS, but usually called "response time").*

`sys Schema` *contains all the necessary info to find that Ugly Duckling:*

```
*************************** 1. row ***************************
     schema_name: ecommerce
         tot_lat: 32.18 s
      exec_count: 199
 latency_per_call: 161.72 ms
query_sample_text: SELECT products.name, COUNT(reviews.rating) AS review_count
FROM ecommerce.products
LEFT JOIN ecommerce.reviews ON products.id = reviews.product_id
GROUP BY products.name
ORDER BY review_count DESC LIMIT 1
```

# Sys Schema is you friend

*The `sys` schema is your friend, it contains all the necessary information to find the queries that need to be optimized.*

*We use these 5 tables containing the necessary information:*

```
SQL> show tables like 'statements_with%';
+---------------------------------------------+
| Tables_in_sys (statements_with%)            |
+---------------------------------------------+
| statements_with_errors_or_warnings          |
| statements_with_full_table_scans            |
| statements_with_runtimes_in_95th_percentile |
| statements_with_sorting                     |
| statements_with_temp_tables                 |
+---------------------------------------------+
5 rows in set (0.0011 sec)
```

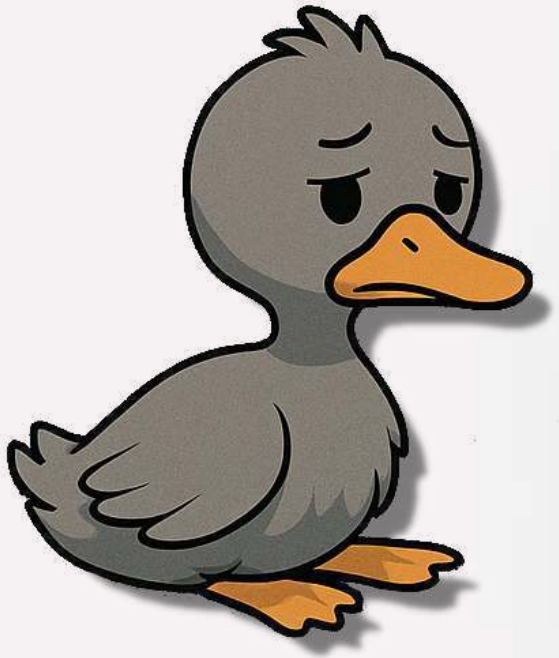
# Recap: know your workload - TIP 7

- *check if your workload is read or write intensive*
- *use* `sys` *schema to find the queries to optimize*
- `QUERY_SAMPLE_TEXT` *contains the eventual comments, so use them!*

```
select name, price /* query 1 */ from products where price>100
```

# Recap: fix the Ugly Duckling - TIP 8

- *regularly check for the query consuming the most of the execution time*



# Query Execution Plan

*how to optimize my queries?*

# Query Execution Plan

*The Query Execution Plan (QEP) is the roadmap that the MySQL Optimizer uses to execute your query.*

*The QEP can be displayed using the keyword* `EXPLAIN`*. There are different formats to display the QEP:*

- *TRADITIONAL*
- *TREE*
- *JSON (v1 and v2)*

# Query Execution Plan

*The Query Execution Plan (QEP) is the roadmap that the MySQL Optimizer uses to execute your query.*

*The QEP can be displayed using the keyword* `EXPLAIN`*. There are different formats to display the QEP:*

- *TRADITIONAL*
- *TREE*
- *JSON (v1 and v2)*

**this is an ESTIMATION on how MySQL would run the query as it is not executed !**

# Query Execution Plan - OUTPUT

*Let's see the QEP for our Ugly Duckling using different formats:*

*We use the following syntax:*

```
EXPLAIN FORMAT=<format>
  SELECT products.name, COUNT(reviews.rating) AS review_count
    FROM ecommerce.products
    LEFT JOIN ecommerce.reviews ON products.id = reviews.product_id
    GROUP BY products.name
    ORDER BY review_count DESC LIMIT 1 \G
```

*If we don't specify the format, the value of the variable* `explain_format` *is used.*

*If* `JSON` *is used, the version is defined by the variable* `explain_json_format_version`.

# Query Execution Plan - TRADITIONAL

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: products
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 79480
     filtered: 100
        Extra: Using temporary; Using filesort
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: reviews
   partitions: NULL
         type: ref
possible_keys: product_id
          key: product_id
      key_len: 5
          ref: ecommerce.products.id
         rows: 1
     filtered: 100
        Extra: NULL
```

# Query Execution Plan - TREE

```
EXPLAIN: -> Limit: 1 row(s)
    -> Sort: review_count DESC, limit input to 1 row(s) per chunk
        -> Table scan on <temporary>
            -> Aggregate using temporary table
                -> Nested loop left join  (cost=51675 rows=124453)
                    -> Table scan on products  (cost=8116 rows=79480)
                    -> Index lookup on reviews using product_id
                          (product_id = products.id)  (cost=0.391 rows=1.57)
```

# Query Execution Plan - JSON


```
EXPLAIN: {
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "51674.79"
    },
    "ordering_operation": {
      "using_filesort": true,
      "grouping_operation": {
        "using_temporary_table": true,
        "using_filesort": false,
        "nested_loop": [
          {
            "table": {
              "table_name": "products",
              "access_type": "ALL",
              "rows_examined_per_scan": 79480,
              "rows_produced_per_join": 79480,
              "filtered": "100.00",
              "cost_info": {
                "read_cost": "168.25",
                "eval_cost": "7948.00",
                "prefix_cost": "8116.25",
                "data_read_per_join": "124M"
              },
              "used_columns": [
                "id",
                "name"
              ]
            }
          },
          {
            "table": {
              "table_name": "reviews",
              "access_type": "ref",
              "possible_keys": [
                "product_id"
              ],
              "key": "product_id",
              "used_key_parts": [
                "product_id"
              ],
              "key_length": "5",
              "ref": [
                "ecommerce.products.id"
              ],
              "rows_examined_per_scan": 1,
              "rows_produced_per_join": 124452,
              "filtered": "100.00",
              "cost_info": {
                "read_cost": "31113.25",
                "eval_cost": "12445.30",
                "prefix_cost": "51674.79",
                "data_read_per_join": "20M"
              },
              "used_columns": [
                "product_id",
                "rating"
              ]
            }
          }
        ]
      }
    }
  }
}
```

# Query Execution Plan - JSONv2


```
EXPLAIN: {
  "query": "/* select#1 */ select `ecommerce`.`products`.`name` AS `name`,count(`ecommerce`.`reviews`.`ratin
g`) AS `review_count` from `ecommerce`.`products` left join `ecommerce`.`reviews` on((`ecommerce`.`reviews`.
`product_id` = `ecommerce`.`products`.`id`)) where true group by `ecommerce`.`products`.`name` order by `rev
iew_count` desc limit 1",
  "query_plan": {
    "limit": 1,
    "inputs": [
      {
        "inputs": [
          {
            "inputs": [
              {
                "inputs": [
                  {
                    "inputs": [
                      {
                        "operation": "Table scan on products",
                        "table_name": "products",
                        "access_type": "table",
                        "schema_name": "ecommerce",
                        "used_columns": [
                          "id",
                          "name"
                        ],
                        "estimated_rows": 79480.0,
                        "estimated_total_cost": 8116.25
                      },
                      {
                        "covering": false,
                        "operation": "Index lookup on reviews using product_id (product_id = products.id)",
                        "index_name": "product_id",
                        "table_name": "reviews",
                        "access_type": "index",
                        "key_columns": [
                          "product_id"
                        ],
                        "schema_name": "ecommerce",
                        "used_columns": [
                          "product_id",
                          "rating"
                        ],
                        "estimated_rows": 1.565840244293213,
                        "lookup_condition": "product_id = products.id",
                        "index_access_type": "index_lookup",
                        "lookup_references": [
                          "ecommerce.products.id"
                        ],
                        "estimated_total_cost": 0.3914620311792975
                      }
                    ],
                    "join_type": "left join",
                    "operation": "Nested loop left join",
                    "access_type": "join",
                    "estimated_rows": 124452.98261642456,
                    "join_algorithm": "nested_loop",
                    "estimated_total_cost": 51674.793915748596
                  }
                ],
                "operation": "Aggregate using temporary table",
                "access_type": "temp_table_aggregate"
              }
            ],
            "operation": "Table scan on <temporary>",
            "table_name": "<temporary>",
            "access_type": "table"
          }
        ],
        "operation": "Sort: review_count DESC, limit input to 1 row(s) per chunk",
        "access_type": "sort",
        "sort_fields": [
          "review_count DESC"
        ],
        "per_chunk_limit": 1
      }
    ],
    "operation": "Limit: 1 row(s)",
    "access_type": "limit",
    "limit_offset": 0
  },
  "query_type": "select",
  "json_schema_version": "2.0"
}
```

# Query Cost

*It's always nice to know the query cost of a query. This is how to obtain it using the* `JSON` *format of* `EXPLAIN`*:*

```
SQL> explain
     format=json into @qep
      SELECT products.name, COUNT(reviews.rating) AS review_count
        FROM ecommerce.products LEFT JOIN ecommerce.reviews
          ON products.id = reviews.product_id
    GROUP BY products.name ORDER BY review_count DESC LIMIT 1;
```

# Query Cost (2)

## V1:

```
SQL>  select json_extract(@qep,
           "$**.query_cost") as query_cost;
+---------------+
| query_cost    |
+---------------+
| ["51674.79"] |
+---------------+
```

## V2:

```
SQL> select max(cast(value as decimal(20,10)))
                          as max_query_cost
      from json_table(
        json_extract(@qep,
              "$**.estimated_total_cost"),
        "$[*]" columns (value json path "$")
      ) as cost_table;
+--------------------+
| max_query_cost     |
+--------------------+
| 51674.7939157486 |
+--------------------+
```

# EXPLAIN ANALYZE - real numbers

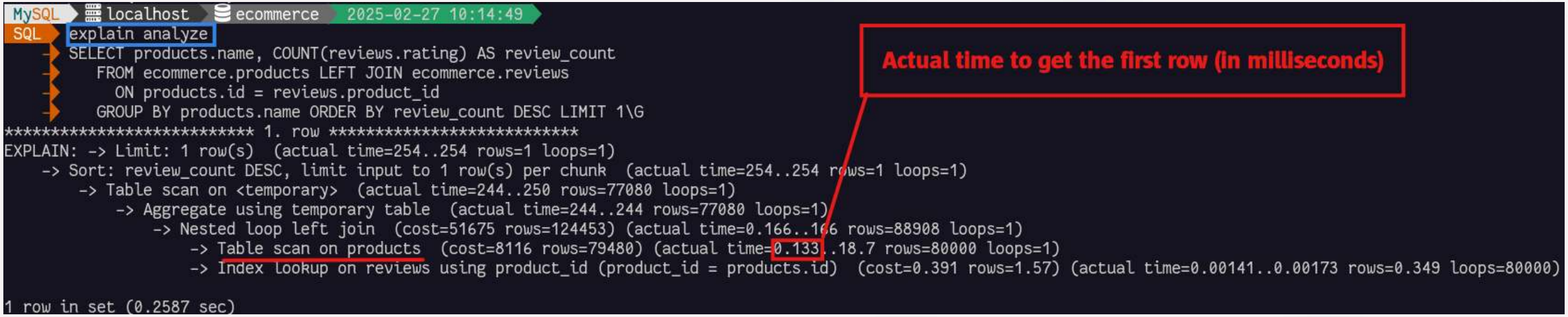
*With* `EXPLAIN ANALYZE` *you can get the real numbers of the query execution:*

```
MySQL  localhost  ecommerce  2025-02-27 10:14:49
SQL  explain analyze
    -> SELECT products.name, COUNT(reviews.rating) AS review_count
    ->    FROM ecommerce.products LEFT JOIN ecommerce.reviews
    ->      ON products.id = reviews.product_id
    ->    GROUP BY products.name ORDER BY review_count DESC LIMIT 1\G
*************************** 1. row ***************************
EXPLAIN: -> Limit: 1 row(s)  (actual time=254..254 rows=1 loops=1)
    -> Sort: review_count DESC, limit input to 1 row(s) per chunk  (actual time=254..254 rows=1 loops=1)
        -> Table scan on <temporary>  (actual time=244..250 rows=77080 loops=1)
            -> Aggregate using temporary table  (actual time=244..244 rows=77080 loops=1)
                -> Nested loop left join  (cost=51675 rows=124453) (actual time=0.166..166 rows=88908 loops=1)
                    -> Table scan on products  (cost=8116 rows=79480) (actual time=0.133..18.7 rows=80000 loops=1)
                    -> Index lookup on reviews using product_id (product_id = products.id)  (cost=0.391 rows=1.57) (actual time=0.00141..0.00173 rows=0.349 loops=80000)

1 row in set (0.2587 sec)
```

# EXPLAIN ANALYZE - real numbers

*With* `EXPLAIN ANALYZE` *you can get the real numbers of the query execution:*



# EXPLAIN ANALYZE - real numbers

*With* `EXPLAIN ANALYZE` *you can get the real numbers of the query execution:*

```
MySQL  localhost  ecommerce  2025-02-27 10:14:49
SQL > explain analyze
    -> SELECT products.name, COUNT(reviews.rating) AS review_count
    ->    FROM ecommerce.products LEFT JOIN ecommerce.reviews
    ->      ON products.id = reviews.product_id
    ->    GROUP BY products.name ORDER BY review_count DESC LIMIT 1\G
*************************** 1. row ***************************
EXPLAIN: -> Limit: 1 row(s)  (actual time=254..254 rows=1 loops=1)
    -> Sort: review_count DESC, limit input to 1 row(s) per chunk  (actual time=254..254 rows=1 loops=1)
        -> Table scan on <temporary>  (actual time=244..250 rows=77080 loops=1)
            -> Aggregate using temporary table  (actual time=244..244 rows=77080 loops=1)
                -> Nested loop left join  (cost=51675 rows=124453) (actual time=0.166..166 rows=88908 loops=1)
                    -> Table scan on products  (cost=8116 rows=79480) (actual time=0.133..18.7 rows=80000 loops=1)
                    -> Index lookup on reviews using product_id (product_id = products.id)  (cost=0.391 rows=1.57) (actual time=0.00141..0.00173 rows=0.349 loops=80000)

1 row in set (0.2587 sec)
```

Actual time to get all rows

# EXPLAIN ANALYZE - real numbers

*With* `EXPLAIN ANALYZE` *you can get the real numbers of the query execution:*

```
MySQL  localhost  ecommerce  2025-02-27 10:14:49
SQL  explain analyze
    -> SELECT products.name, COUNT(reviews.rating) AS review_count
    ->    FROM ecommerce.products LEFT JOIN ecommerce.reviews
    ->      ON products.id = reviews.product_id
    ->    GROUP BY products.name ORDER BY review_count DESC LIMIT 1\G
*************************** 1. row ***************************
EXPLAIN: -> Limit: 1 row(s)  (actual time=254..254 rows=1 loops=1)
    -> Sort: review_count DESC, limit input to 1 row(s) per chunk  (actual time=254..254 rows=1 loops=1)
        -> Table scan on <temporary>  (actual time=244..250 rows=77080 loops=1)
            -> Aggregate using temporary table  (actual time=244..244 rows=77080 loops=1)
                -> Nested loop left join  (cost=51675 rows=124453) (actual time=0.166..166 rows=88905 loops=1)
                    -> Table scan on products  (cost=8116 rows=79480) (actual time=0.133..18.7 rows=80000 loops=1)
                    -> Index lookup on reviews using product_id (product_id = products.id)  (cost=0.391 rows=1.57) (actual time=0.00141..0.00173 rows=0.349 loops=80000)

1 row in set (0.2587 sec)
```

Actual number of rows read

# EXPLAIN ANALYZE - real numbers

*With* `EXPLAIN ANALYZE` *you can get the real numbers of the query execution:*

```
MySQL  localhost  ecommerce  2025-02-27 10:14:49
SQL > explain analyze
    -> SELECT products.name, COUNT(reviews.rating) AS review_count
    ->    FROM ecommerce.products LEFT JOIN ecommerce.reviews
    ->      ON products.id = reviews.product_id
    ->    GROUP BY products.name ORDER BY review_count DESC LIMIT 1\G
*************************** 1. row ***************************
EXPLAIN: -> Limit: 1 row(s)  (actual time=254..254 rows=1 loops=1)
    -> Sort: review_count DESC, limit input to 1 row(s) per chunk  (actual time=254..254 rows=1 loops=1)
        -> Table scan on <temporary>  (actual time=244..250 rows=77080 loops=1)
            -> Aggregate using temporary table  (actual time=244..244 rows=77080 loops=1)
                -> Nested loop left join  (cost=51675 rows=124453) (actual time=0.166..166 rows=88908 loops=1)
                    -> Table scan on products  (cost=8116 rows=79480) (actual time=0.133..18.7 rows=80000 loops=1)
                    -> Index lookup on reviews using product_id (product_id = products.id)  (cost=0.391 rows=1.57) (actual time=0.00141..0.00173 rows=0.349 loops=80000)

1 row in set (0.2587 sec)
```

Actual number of loops

# Old DBA verification

*We can verify this by using this simple test:*

```
SQL> flush status;
SQL> select products.name, count(reviews.rating) as review_count
     from ecommerce.products
     left join ecommerce.reviews on products.id = reviews.product_id
     group by products.name order by review_count desc limit 1;
SQL> show status like 'handler_read_%';
+-----------------------+--------+
| Variable_name         | Value  |
+-----------------------+--------+
| Handler_read_first    | 1      |
| Handler_read_key      | 168909 |
| Handler_read_last     | 0      |
| Handler_read_next     | 27943  |
| Handler_read_prev     | 0      |
| Handler_read_rnd      | 0      |
| Handler_read_rnd_next | 157082 |
+-----------------------+--------+
```

# Old DBA verification

*We can verify this by using this simple test:*

```
SQL> flush status;
SQL> select products.name, count(reviews.rating) as review_count
     from ecommerce.products
     left join ecommerce.reviews on products.id = reviews.product_id
     group by products.name order by review_count desc limit 1;
SQL> show status like 'handler_read_%';
+-----------------------+--------+
```

```
SQL> select 77080+80000;
+--------------+
| 77080+80000 |
+--------------+
|       157080 |
+--------------+
```

```
+-----------------------+--------+
```

# Optimizer Traces

*For those who want to go deeper, the MySQL Optimizer provides a trace feature.*

```
SQL> set optimizer_trace="enabled=on";
SQL> explain format=tree <your query>;
SQL> select * from information_schema.optimizer_trace\G

                              ... ENJOY ;-) ...

SQL> set optimizer_trace="enabled=off";
```

# Let's try to optimize our Ugly Duckling

*We can see that the query is using a temporary table and a filesort, and it starts by performing a full table scan on the* `products` *table.*

*The first think to do is to check both tables:*

```
SQL> desc products;
+-------------+--------------+------+-----+-------------------+-------------------+
| Field       | Type         | Null | Key | Default           | Extra             |
+-------------+--------------+------+-----+-------------------+-------------------+
| name        | varchar(255) | YES  |     | NULL              |                   |
| description | varchar(150) | YES  |     | NULL              |                   |
| price       | float        | YES  |     | NULL              |                   |
| created_at  | datetime     | YES  |     | CURRENT_TIMESTAMP | DEFAULT_GENERATED |
+-------------+--------------+------+-----+-------------------+-------------------+
```

# Let's try to optimize our Ugly Duckling (2)

```
SQL> desc reviews;
+------------+--------------+------+-----+-------------------+-------------------+
| Field      | Type         | Null | Key | Default           | Extra             |
+------------+--------------+------+-----+-------------------+-------------------+
| user_id    | varchar(36)  | YES  | MUL | NULL              |                   |
| product_id | int          | YES  | MUL | NULL              |                   |
| rating     | int          | YES  |     | NULL              |                   |
| review     | text         | YES  |     | NULL              |                   |
| created_at | datetime     | YES  |     | CURRENT_TIMESTAMP | DEFAULT_GENERATED |
| id         | int unsigned | NO   | PRI | NULL              | auto_increment    |
+------------+--------------+------+-----+-------------------+-------------------+
```

# Let's try to optimize our Ugly Duckling (2)

```
SQL> desc reviews;
+------------+--------------+------+-----+-------------------+-------------------+
| Field      | Type         | Null | Key | Default           | Extra             |
+------------+--------------+------+-----+-------------------+-------------------+
| user_id    | varchar(36)  | YES  | MUL | NULL              |                   |
| product_id | int          | YES  | MUL | NULL              |                   |
| rating     | int          | YES  |     | NULL              |                   |
| review     | text         | YES  |     | NULL              |                   |
```

*Remember the query is:*

```
SELECT products.name, COUNT(reviews.rating) AS review_count
  FROM ecommerce.products LEFT JOIN ecommerce.reviews
    ON products.id = reviews.product_id
  GROUP BY products.name ORDER BY review_count DESC LIMIT 1\G
```

*And the cost was* `51674.7939157486`

# Let's try to optimize our Ugly Duckling (3)

*Then we realize we could replace the* `LEFT JOIN` *by a* `JOIN`*:*

```
SQL> SELECT products.name, COUNT(reviews.rating) AS review_count
       FROM ecommerce.products JOIN ecommerce.reviews
         ON products.id = reviews.product_id
         GROUP BY products.name ORDER BY review_count DESC LIMIT 1\G
```

```
EXPLAIN: -> Limit: 1 row(s)
    -> Sort: review_count DESC, limit input to 1 row(s) per chunk
        -> Table scan on <temporary>
            -> Aggregate using temporary table
                -> Nested loop inner join  (cost=12519 rows=27588)
                    -> Filter: (reviews.product_id is not null)  (cost=2863 rows=27588)
                        -> Table scan on reviews  (cost=2863 rows=27588)
                    -> Single-row index lookup on products using PRIMARY (id = reviews.product_id)  (cost=0.25 rows=1)
```

# Let's try to optimize our Ugly Duckling (3)

*Then we realize we could replace the* `LEFT JOIN` *by a* `JOIN`*:*

```
SQL> SELECT products.name, COUNT(reviews.rating) AS review_count
       FROM ecommerce.products JOIN ecommerce.reviews
         ON products.id = reviews.product_id
         GROUP BY products.name ORDER BY review_count DESC LIMIT 1\G
```

*We have now the following cost:*

```
+------------------+
| max_query_cost   |
+------------------+
| 12518.8500000000 |
+------------------+
```

# Let's try to optimize our Ugly Duckling

# better is the enemy of good

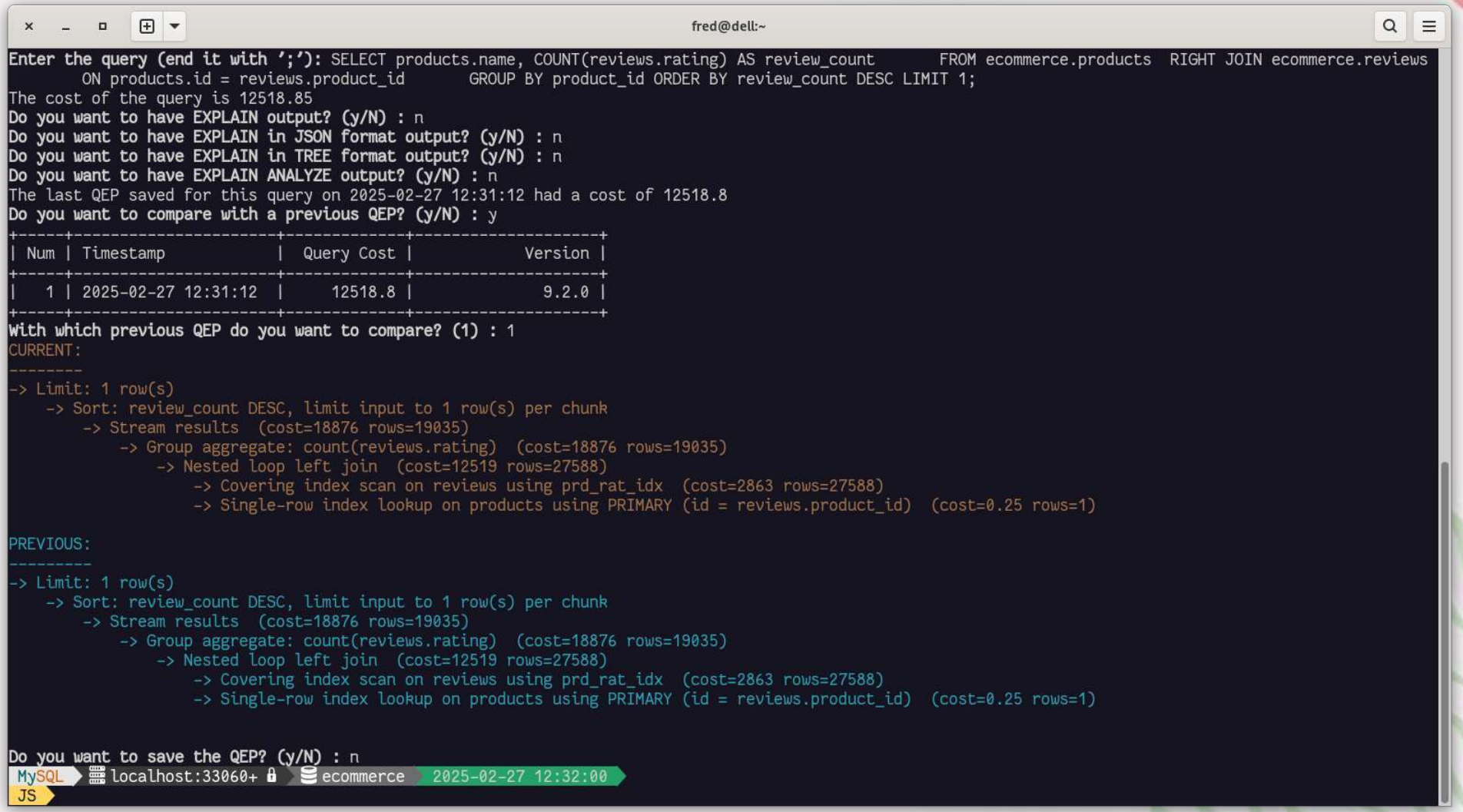
*We could have created an index on* `(product_id, rating)` *in the* `reviews` *table and group by* `product_id`*, but the cost would have been higher:*

```
EXPLAIN: -> Limit: 1 row(s)
    -> Sort: review_count DESC, limit input to 1 row(s) per chunk
        -> Stream results  (cost=18876 rows=19035)
            -> Group aggregate: count(reviews.rating)  (cost=18876 rows=19035)
                -> Nested loop inner join  (cost=12519 rows=27588)
                    -> Filter: (reviews.product_id is not null)  (cost=2863 rows=27588)
                        -> Covering index scan on reviews using prd_rat_idx  (cost=2863 rows=27588)
                    -> Single-row index lookup on products using PRIMARY (id = reviews.product_id)
                                                                        (cost=0.25 rows=1)
```

*The cost is now* `18875.5320276498`

# Recap: optimize your queries - TIP 9

- *use* `EXPLAIN` *to get the Query Execution Plan*
- *use* `EXPLAIN ANALYZE` *to get the real numbers*
- *compare the costs of different queries*
- *verify QEP and cost over time (I encourage you to save them)*
  - *use* `statement_digest_text()`
  - *trim the output and hash it to save the query*
    - `SHA2(TRIM(statement_digest_text(<query>)), 224)`



# Split your workload

*reads and writes*

# Best Practices: split your workload

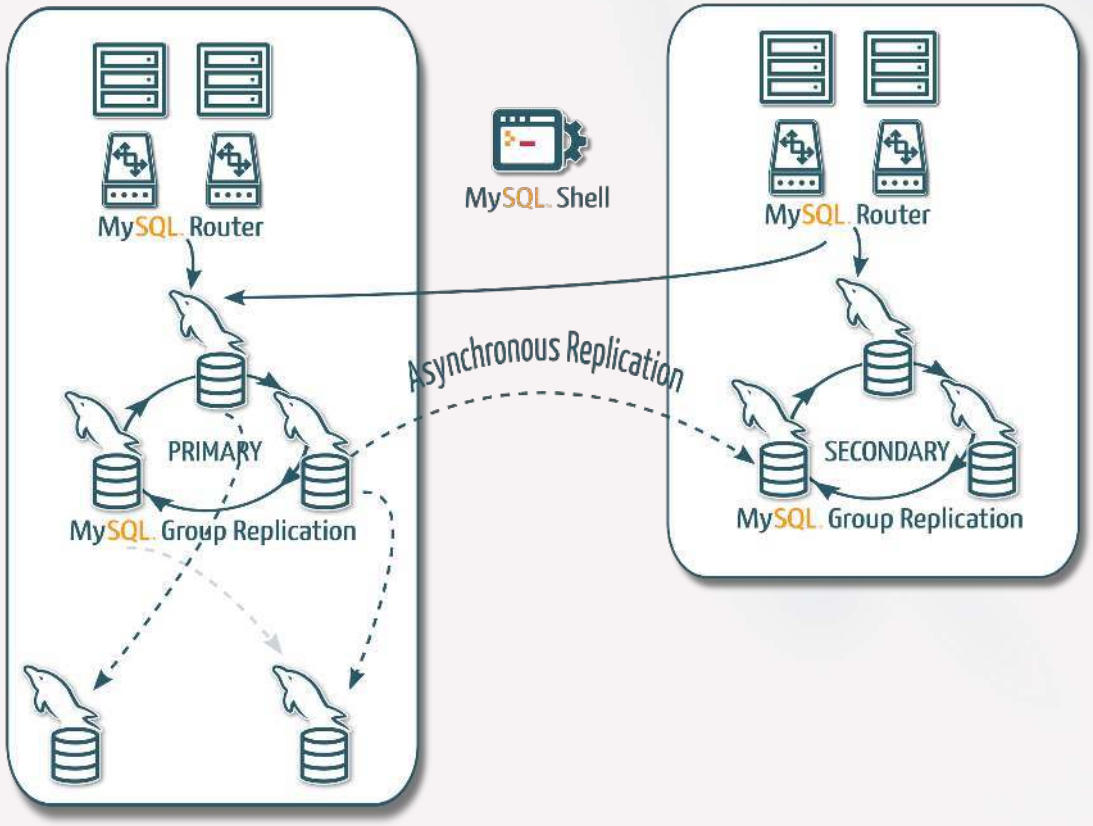
*It's always a good idea to split your workload between reads and writes using dedicated connections.*

*Even if you are using the same server to start with. Such configuration will allow you to scale your infrastructure more easily and the DBAs and/or OPS will be very happy.*

```
conn_r = DriverManager.getConnection("jdbc:mysql://localhost/ecommerce?" +
                                     "user=reader&password=");
conn_w = DriverManager.getConnection("jdbc:mysql://localhost/ecommerce?" +
                                     "user=writer&password=");
```

# Best Practices: split your workload (2)

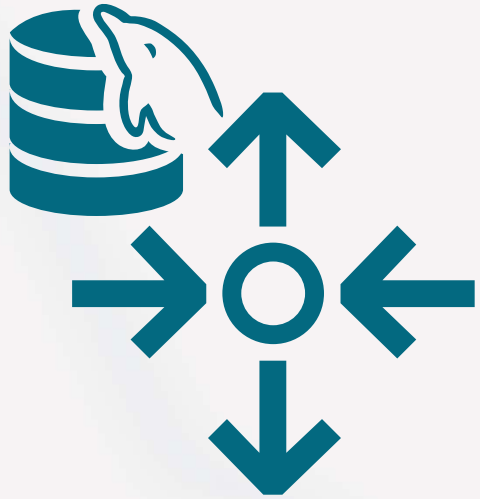
*You can also use different servers for reads and writes and then you will be prepared to scale your infrastructure like this:*



# Best Practices: split your workload (3)

*Some connectors, including MySQL Connector/J support multi-host connections.*

*Or you can use MySQL Router to manage the read/write split transparently for you.*



# Recap: prepare your infrastructure for scaling - TIP 10

- *split your workload between reads and writes*
- *use different servers for reads and writes*
- *use MySQL Router to manage the read/write split*

# MySQL & InnoDB

*the secret variables making all the difference*

# DBA's job

*Tunning InnoDB is more a DBA's job than a developer's job. But aren't we speaking about "full stack developers" ?*

*So let me share some secrets with you to make your application faster.*

# DBA's job

*Tunning InnoDB is more a DBA's job than a developer's job. But aren't we speaking about "full stack developers" ?*

*So let me share some secrets with you to make your application faster.*

## Secret #1:

# DBA's job

*Tunning InnoDB is more a DBA's job than a developer's job. But aren't we speaking about "full stack developers" ?*

*So let me share some secrets with you to make your application faster.*

## Secret #1:

## memory is fast!

# InnoDB Buffer Pool

*It's important to have the Working Set in memory !*

*Less your read from disk faster your queries will be.*

# InnoDB Buffer Pool

*It's important to have the Working Set in memory !*

*Less your read from disk faster your queries will be.*

*We can verify that most of the page requests are coming from memory:*

```
SQL> show global status like 'innodb_buffer_pool_read%s';
+---------------------------------+------------+
| Variable_name                   | Value      |
+---------------------------------+------------+
| Innodb_buffer_pool_read_requests | 1201291089 |
| Innodb_buffer_pool_reads         | 1986       |
+---------------------------------+------------+
2 rows in set (0.0069 sec)
```

# InnoDB Buffer Pool (2)

*We need to keep the ratio between pages requested and pages read from disk as low as possible.*

```
SQL> select concat(format(B.num * 100.0 / A.num,2),'%') DiskReadRatio
     from (
       select variable_value num from performance_schema.global_status
        where variable_name = 'Innodb_buffer_pool_read_requests') A,
     ( select variable_value num from performance_schema.global_status
        where variable_name = 'Innodb_buffer_pool_reads') B;
+---------------+
| DiskReadRatio |
+---------------+
| 0.00%         |
+---------------+
1 row in set (0.0008 sec)
```

# InnoDB Buffer Pool (3)

*If the ratio is above 10%, I would recomment to increase the size of the Buffer Pool and/or check if less pages could be read from disk by reducing the working set (less full table scans, archiving old data, using partitioning, etc).*

*Check what's in the Buffer Pool, you might be surprised:*

```
SQL> SELECT TABLE_NAME,INDEX_NAME, COUNT(*) AS Pages,
      ROUND(SUM(IF(COMPRESSED_SIZE = 0, 16384, COMPRESSED_SIZE))/1024/1024)
         AS 'Total Data (MB)'
      FROM INFORMATION_SCHEMA.INNODB_BUFFER_PAGE
      WHERE table_name not like '`mysql`.%'
      GROUP BY TABLE_NAME,INDEX_NAME order by 4 desc,3 desc;
```

# InnoDB Buffer Pool (3)

*We can verify the Buffer Pool's usage with this query:*

```
SQL> SELECT format_bytes(@@innodb_buffer_pool_size) BufferPoolSize,
        FORMAT(A.num * 100.0 / B.num,2) BufferPoolFullPct,
        FORMAT(C.num * 100.0 / D.num,2) BufferPollDirtyPct
FROM
    (SELECT variable_value num FROM performance_schema.global_status
    WHERE variable_name = 'Innodb_buffer_pool_pages_data') A,
    (SELECT variable_value num FROM performance_schema.global_status
    WHERE variable_name = 'Innodb_buffer_pool_pages_total') B,
    (SELECT variable_value num FROM performance_schema.global_status
    WHERE variable_name='Innodb_buffer_pool_pages_dirty') C,
    (SELECT variable_value num FROM performance_schema.global_status
    WHERE variable_name='Innodb_buffer_pool_pages_total') D;
```

# InnoDB Buffer Pool (3)

*We can verify the Buffer Pool's usage with this query:*

```
SQL> SELECT format_bytes(@@innodb_buffer_pool_size) BufferPoolSize,
        FORMAT(A.num * 100.0 / B.num,2) BufferPoolFullPct,
        FORMAT(C.num * 100.0 / D.num,2) BufferPollDirtyPct
FROM
    (SELECT variable_value num FROM performance_schema.global_status
    WHERE variable_name = 'Innodb_buffer_pool_pages_data') A,
```

```
+----------------+--------------------+--------------------+
| BufferPoolSize | BufferPoolFullPct | BufferPollDirtyPct |
+----------------+--------------------+--------------------+
| 128.00 MiB     | 69.10              | 8.17               |
+----------------+--------------------+--------------------+
```

# InnoDB Buffer Pool - Warm Buffer Pool

*As developer, you might start a MySQL instance, execute some queries and then stop the instance. But the performance of such queries could be not optimal as the Buffer Pool is cold.*

*MySQL provides a solution for that.*

# InnoDB Buffer Pool - Warm Buffer Pool (2)

*To always start with a warm Buffer Pool, you can dump the content of the InnoDB Buffer Pool to disk and load it at startup:*

```
SQL> set persist innodb_buffer_pool_dump_at_shutdown = 1;
SQL> set persist innodb_buffer_pool_load_at_startup = 1;
```

# InnoDB Buffer Pool - Warm Buffer Pool (2)

*To always start with a warm Buffer Pool, you can dump the content of the InnoDB Buffer Pool to disk and load it at startup:*

```
SQL> set persist innodb_buffer_pool_dump_at_shutdown = 1;
SQL> set persist innodb_buffer_pool_load_at_startup = 1;
```

*In production, I would recommend to dump the content of the BP at regular intervals has the working set could change quickly over time.*

```
SQL> create event automatic_bufferpool_dump
       on schedule every 1 hour
        do
         set global innodb_buffer_pool_dump_now=on;
```

# InnoDB Redo Log Capacity

*If your database is write intensive, having a correct redo log capacity is important to avoid stalls.*

*The rule of thumb is to have a redo log capacity of 1 hour to allow InnoDB's checkpointing to run smoothly.*

*This setting is controlled by the variable* `innodb_log_capacity`.

# InnoDB Redo Log Capacity (2)

*The best way to find the adequate value is to run the following query (on single line) at peak time:*

```
SQL> select variable_value from performance_schema.global_status
    where variable_name='innodb_redo_log_current_lsn' into @a;select
    sleep(60) into @garb; select variable_value from performance_schema.global_status
    where variable_name='innodb_redo_log_current_lsn' into @b;select
    format_bytes(abs(@a - @b)) per_min, format_bytes(abs(@a - @b)*60) per_hour;
Query OK, 1 row affected (0.0005 sec)
Query OK, 1 row affected (1 min 0.0002 sec)
Query OK, 1 row affected (0.0006 sec)
+----------+------------+
| per_min  | per_hour   |
+----------+------------+
| 5.69 MiB | 341.11 MiB |
+----------+------------+
1 row in set (0.0006 sec)
```

# InnoDB Redo Log Capacity (2)

*The best way to find the adequate value is to run the following query (on single line) at peak time:*

```
SQL> select variable_value from performance_schema.global_status
    where variable_name='innodb_redo_log_current_lsn' into @a;select
    sleep(60) into @garb; select variable_value from performance_schema.global_status
    where variable_name='innodb_redo_log_current_lsn' into @b;select
    format_bytes(abs(@a - @b)) per_min, format_bytes(abs(@a - @b)*60) per_hour;
```

```
SQL> set persist innodb_redo_log_capacity=350*1024*1024;
```

```
+----------+------------+
| per_min  | per_hour   |
+----------+------------+
| 5.69 MiB | 341.11 MiB |
+----------+------------+
1 row in set (0.0006 sec)
```

# Recap: keep your working set in memory - TIP 11

- *check the Buffer Pool usage*
- *keep the ratio between pages requested and pages read from disk as low as possible*

# Recap: use a Buffer Pool always warm - TIP 12

- *dump the content of the Buffer Pool at shutdown*
- *dump regularly the content of the Buffer Pool using an event*
- *load the content of the Buffer Pool at startup*

# Recap: have a correct redo log capacity - TIP 13

- *check the redo log capacity*
- *set the redo log capacity to 1 hour*

# Recap: let MySQL setup InnoDB for you - TIP 14

*On a dedicated MySQL Server, the best is to let InnoDB decide the size of the Buffer Pool and the Redo Log Capacity.*

*In* `my.cnf`*:*

```
innodb_dedicated_server=1
```

*See https://dev.mysql.com/doc/refman/8.0/en/innodb-dedicated-server.html*

# Add some limits

*to avoid bad surprises*

# Cap the Query Time

*It's possible to stop the execution of a query,* `SELECT`*(*), if it takes too long.*

*The value of "too long" is defined in the variable* `max_execution_time` *or using an optimizer hint:*

```
select /*+ max_execution_time(5000) */ sleep(10);
+-----------+
| sleep(10) |
+-----------+
|         1 |
+-----------+
1 row in set (5.0006 sec)
```

*(*) not part of a store procedure*

# Cap the amount of returned rows

*Of course you can use the* `LIMIT` *keyword to limit the amount of rows returned by a query.*

*But it's also possible to limit the amount of rows returned by a query using a variable (session or global) to avoid bad surprises.*

# Cap the amount of returned rows (2)

```
SQL> select count(*) from orders;
+----------+
| count(*) |
+----------+
|     3359 |
+----------+

SQL> set sql_select_limit=5;

SQL> select * from orders;
+----+--------------------------------------+-------------+---------------------+
| id | user_id                              | total_price | created_at          |
+----+--------------------------------------+-------------+---------------------+
|  1 | a2293479-f35a-11ef-9704-5e1b9081e705 |      841.21 | 2025-02-25 10:27:54 |
|  2 | a21e7b98-f35a-11ef-9704-5e1b9081e705 |      417.41 | 2025-02-25 10:27:54 |
|  3 | a3e980a6-f35a-11ef-9704-5e1b9081e705 |     2163.46 | 2025-02-25 10:27:54 |
|  4 | 9f4b2ef2-f35a-11ef-9704-5e1b9081e705 |     1640.24 | 2025-02-25 10:27:54 |
|  5 | a23e395c-f35a-11ef-9704-5e1b9081e705 |     3090.51 | 2025-02-25 10:27:54 |
+----+--------------------------------------+-------------+---------------------+
5 rows in set (0.0003 sec)
```

# Connections Tracking and Limiting

*To avoid bad surprises (like swapping), it's possible to track and limit the memory consumption of the connections.*

*To enable it you need to set* `global_connection_memory_tracking` *to* `1`*:*

```
SQL> set global global_connection_memory_tracking=1;
```

# Connections Tracking and Limiting

*To avoid bad surprises (like swapping), it's possible to track and limit the memory consumption of the connections.*

*To enable it you need to set* `global_connection_memory_tracking` *to* `1`*:*

```
SQL> set global global_connection_memory_tracking=1;
```

*You can limit the connection memory limit:*

```
SQL> set <global/session> connection_memory_limit=2200000;
SQL> set global global_connection_memory_limit=536870912000;
```

# Connections Tracking and Limiting (2)

*To know the Global Connection Consumption Memory:*

```
SQL> select format_bytes(variable_value) global_connection_memory
        from performance_schema.global_status
        where variable_name='global_connection_memory';
+--------------------------+
| global_connection_memory |
+--------------------------+
| 16.22 MiB                |
+--------------------------+
```

# Connections Tracking and Limiting (3)

*If the limit it reached, the user will be disconnected with the following error:*

```
ERROR: 4081 (HY000): Connection closed. Global connection memory limit 16777216
bytes exceeded. Consumed 16949968 bytes.
```

*This limitation doesn't appy to users with* `CONNECTION_ADMIN` *privilege.*

# Don't wait

*It's possible to skip to wait for timeout when a lock is set on a row.*

*Depending on your use case, you can decide to not wait or skip the locked rows.*

*Let's see how to do that.*

*In one session, we do:*

```
Session 1 SQL> start transaction;
Session 1 SQL> update reviews set rating=rating+1  where product_id=26719;
Query OK, 7 rows affected (0.0004 sec)
```

# Don't wait (2)

*This is the usual behaviour when a lock is set on row(s):*

```
Session 2 SQL> select user_id, rating from reviews where product_id=26719 for update ;
ERROR: 1205 (HY000): Lock wait timeout exceeded; try restarting transaction

Session 2 SQL> show variables like 'innodb_lock_wait_timeout';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| innodb_lock_wait_timeout | 50    |
+---------------------------+-------+
1 row in set (0.0021 sec)
```

# Don't wait (3)

*Now let's try to use* `NOWAIT` *and* `SKIP LOCKED` *in two different sessions:*

```
Session 2 SQL> select user_id, rating
                 from reviews
                where product_id=26719
                  for update nowait;
ERROR: 3572 (HY000): Statement aborted because
lock(s) could not be acquired immediately and
NOWAIT is set.
```

# Don't wait (3)

*Now let's try to use* `NOWAIT` *and* `SKIP LOCKED` *in two different sessions:*

```
Session 2 SQL> select user_id, rating
                 from reviews
                where product_id=26719
                  for update nowait;
ERROR: 3572 (HY000): Statement aborted because
lock(s) could not be acquired immediately and
NOWAIT is set.
```

```
Session 3 SQL> select user_id, rating
                 from reviews
                where product_id=26719
                  for update skip locked;
Empty set (0.0008 sec)
```

# Recap: fix limits - TIP 15

- *cap the query time*
- *cap the amount of returned rows*
- *limit the memory consumption of the connections*
- *don't wait for locks*

# Memory Allocator

*avoid glibc*

# Memory Allocator

*The default memory allocator in Linux distribution (`glibc-malloc`) doesn't perform well in high concurrency environments and should be avoided !*

*This let us with 2 choices:*

- *jemalloc (good for perf, but less RAM management efficiency)*
- *tcmalloc (recommended choice)*

# Memory Allocator (2)

*Install* `tcmalloc`*:*

```
$ sudo yum -y install gperftools-libs
```

*And in* `systemd` *service file you need to add:*

```
$ sudo EDITOR=vi systemctl edit mysqld
[Service]
Environment="LD_PRELOAD=/usr/lib64/libtcmalloc_minimal.so.4"
```

# Memory Allocator (3)

*Reload the service and restart MySQL:*

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart mysqld
```

# Memory Allocator (3)

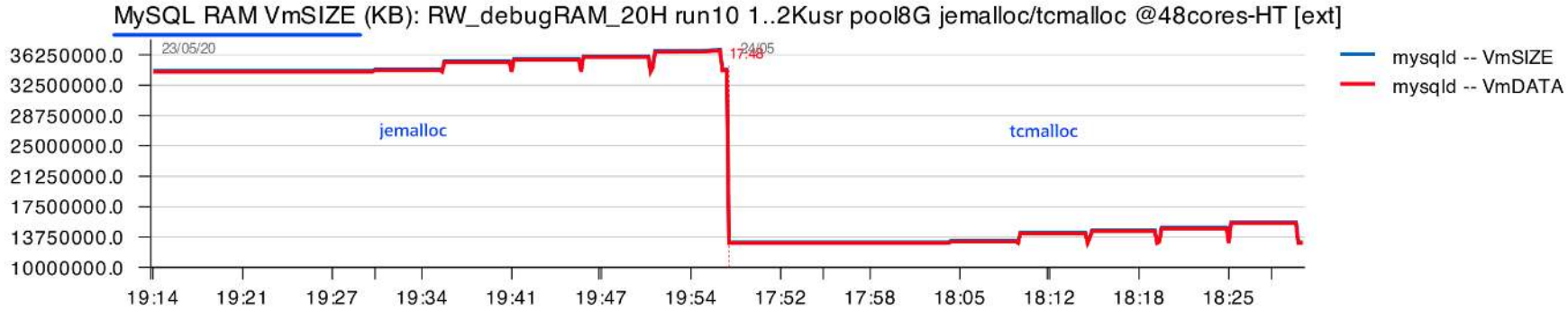
*Reload the service and restart MySQL:*

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart mysqld
```

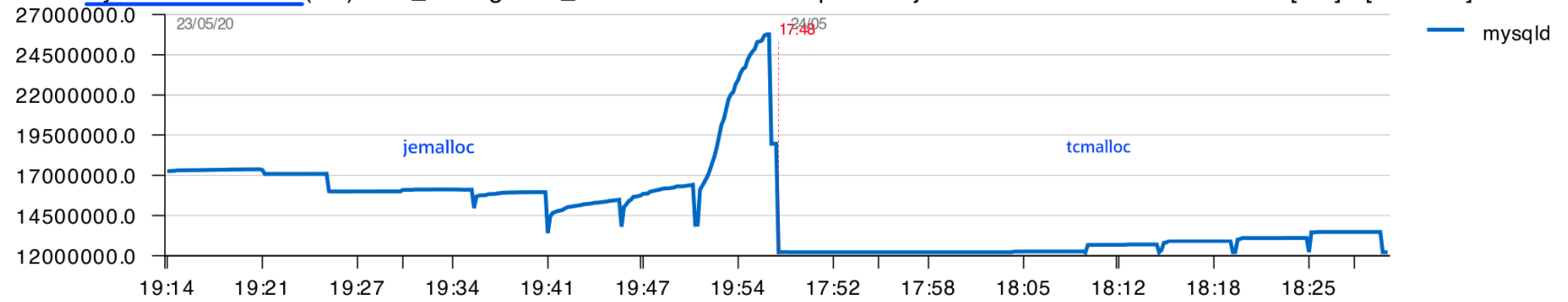
*You can verify that* **`mysqld`** *is using the new memory allocator with* **`pldd`**:

```
$ sudo pldd $(pidof mysqld)
22738:    /usr/sbin/mysqld
linux-vdso.so.1
/usr/lib64/libtcmalloc_minimal.so.4
/lib64/libpthread.so.0
/usr/lib64/mysql/private/libprotobuf-lite.so.3.11.4
/lib64/libcrypt.so.2
/lib64/librt.so.1
/lib64/libcrypto.so.1.1
...
```

# Memory Allocator: jemalloc vs tcmalloc:




VmSize: sum of all mapped memory
VmData: size of data, stack, and text segments




vmRSS: size of memory portions (Resident Set Size)

courtesy of DimitriK

# Recap: memory allocator - TIP 16

*Don't use* `glibc-malloc` *as memory allocator for MySQL !*

- *use* `tcmalloc` *(recommended)*
- *use* `jemalloc` *(good for performance, but less RAM management efficiency)*

# Swap

*Your enemy*

# Memory - Swap

*As a MySQL DBA, you should hate when your server is swapping and you are right ! Using the swap should be avoided at any cost.*

# Memory - Swap

*As a MySQL DBA, you should hate when your server is swapping and you are right ! Using the swap should be avoided at any cost.*

*However, disabling the swap is also a bad idea, it's better to swap than to have MySQL killed !*

# Memory - Swap

*As a MySQL DBA, you should hate when your server is swapping and you are right ! Using the swap should be avoided at any cost.*

*However, disabling the swap is also a bad idea, it's better to swap than to have MySQL killed !*

*But we need to reduce the swap usage as much as possible.*

*Let's verify if the server uses the swap:*

```
# free -h
              total        used        free      shared  buff/cache   available
Mem:           15G        5.2G        3.5G       105M        6.8G        9.8G
Swap:         4.0G         24M        4.0G
```

# Memory - Swap (2)

*Is* `mysqld` *using the swap ?*

# Memory - Swap (2)

*Is* `mysqld` *using the swap ?*

```
# cat /proc/$(pidof mysqld)/status | grep Swap
VmSwap:         0 kB
```

*In case it's MySQL using the swap, you can also verify how often is the swap used using* `vmstat 1 10`.

# Memory - Swap (2)

*Is* **`mysqld`** *using the swap ?*

```
# cat /proc/$(pidof mysqld)/status | grep Swap
VmSwap:         0 kB
```

*In case it's MySQL using the swap, you can also verify how often is the swap used using* **`vmstat 1 10`**.

*On Linux, it's the* **`swappiness`** *that controls the tendency of the kernel to move out of physical memory to the swap.*

# Memory - Swap (3)

*The default value(60) is way too high for a decicated MySQL server and should be reduced. Don't set it to 0 (which was recommended in kernels < 2.6.32).*

*I recommend to set the swappiness to a value between 1 and 5:*

```
# systctl -w vm.swappiness=1
```

# Memory - Swap (4)

*NUMA can also be a reason of swapping, so if you have a NUMA system, you should also check the* `numa_balancing` *setting.*

*The question you need to answer is : does my CPU have multiple NUMA cores ?*

# Memory - Swap (4)

*NUMA can also be a reason of swapping, so if you have a NUMA system, you should also check the* **`numa_balancing`** *setting.*

*The question you need to answer is : does my CPU have multiple NUMA cores ?*

*You can provide an answer using* **`numactl`** *command on Linux:*

```
# numactl -H | grep available
available: 4 nodes (0-3)
```

*If the result is > 1, then you would benefit from using:*

```
innodb_numa_interleave = 1
```

*(*) enabled by default when the system supports it*

# Recap: avoid swapping - TIP 17

*Your system should have swap enabled but it should not use it.*

- *swappiness should be set to a low value (1-5)*
- `mysqld` *should not use the swap*
- *if you have a NUMA system, set* `innodb_numa_interleave = 1`

# Adaptive Hash Index

*really useful?*

# Adaptive Hash Index

*When some secondary indexes values are being accessed very frequently, InnoDB builds a hash index (AHI) for them in memory on top of the B-Tree indexes.*

*If your workload doesn't benefit from it, you are disadvantaged by its overhead.*

*This can detected by seeing a lot of waits on rw-lock semaphores in the* `btr0sea.cc` *file (we will cover that later).*

*Some information is available in the* `SHOW ENGINE INNODB STATUS` *output, however in MySQL 8.0 we tend to replace the use of that statement using* `Performance_Schema` *and* `Sys`.

# Adaptive Hash Index

```
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf: size 1, free list len 0, seg size 2, 0 merges
merged operations:
 insert 0, delete mark 0, delete 0
discarded operations:
 insert 0, delete mark 0, delete 0
Hash table size 664177, node heap has 0 buffer(s)
Hash table size 664177, node heap has 0 buffer(s)
Hash table size 664177, node heap has 0 buffer(s)
Hash table size 664177, node heap has 0 buffer(s)
Hash table size 664177, node heap has 0 buffer(s)
Hash table size 664177, node heap has 0 buffer(s)
Hash table size 664177, node heap has 0 buffer(s)
Hash table size 664177, node heap has 0 buffer(s)
0.00 hash searches/s, 8786.34 non-hash searches/s
```

# Adaptive Hash Index (2)

*Statistics for AHI are also available when InnoDB monitor is enabled.*

```
SQL> SELECT Variable_name, Variable_value FROM sys.metrics
     WHERE Variable_name LIKE 'adaptive%';
+---------------------------------------+----------------+
| Variable_name                         | Variable_value |
+---------------------------------------+----------------+
| adaptive_hash_pages_added             | 0              |
| adaptive_hash_pages_removed           | 0              |
| adaptive_hash_rows_added              | 0              |
| adaptive_hash_rows_deleted_no_hash_entry | 0           |
| adaptive_hash_rows_removed            | 0              |
| adaptive_hash_rows_updated            | 0              |
| adaptive_hash_searches                | 1243           |
| adaptive_hash_searches_btree          | 20125723       |
+---------------------------------------+----------------+
```

# Adaptive Hash Index (2)

```
SELECT CONCAT(
        ROUND(
           (
             SELECT Variable_value FROM sys.metrics
             WHERE Variable_name = 'adaptive_hash_searches'
           ) /
           (
             (
              SELECT Variable_value FROM sys.metrics
              WHERE Variable_name = 'adaptive_hash_searches_btree'
             )  + (
              SELECT Variable_value FROM sys.metrics
              WHERE Variable_name = 'adaptive_hash_searches'
             )
           ) * 100,2
         ),'%') 'AHI ratio';
+-----------+
| AHI ratio |
+-----------+
| 0.01%     |
+-----------+
```

# Adaptive Hash Index (3)

*If you don't benefit from AHI, you should disable it.*

```
innodb_adaptive_hash_index = 0
```

# Adaptive Hash Index (3)

*If you don't benefit from AHI, you should disable it.*

```
innodb_adaptive_hash_index = 0
```

*If you benefit from it but you see wrong distribution or many hash partition with high numbers, you should then change the amount of partitions: (max: 512)*

```
innodb_adaptive_hash_index_parts = 8
```

*Since MySQL 8.4, the adaptive hash index is disabled by default.*

# Recap: AHI - TIP 18

*Most of the time, the Adaptive Hash Index is not useful and should be disabled.*

*You will also benefit from the the new* `DROP TABLE|TABLESACE` *and* `TRUNCATE` *improvements when AHI is disabled.*

# Optimize Binlogs

*How to improve performance, reduce disk space and improve asynchonous replication*

# Binlogs

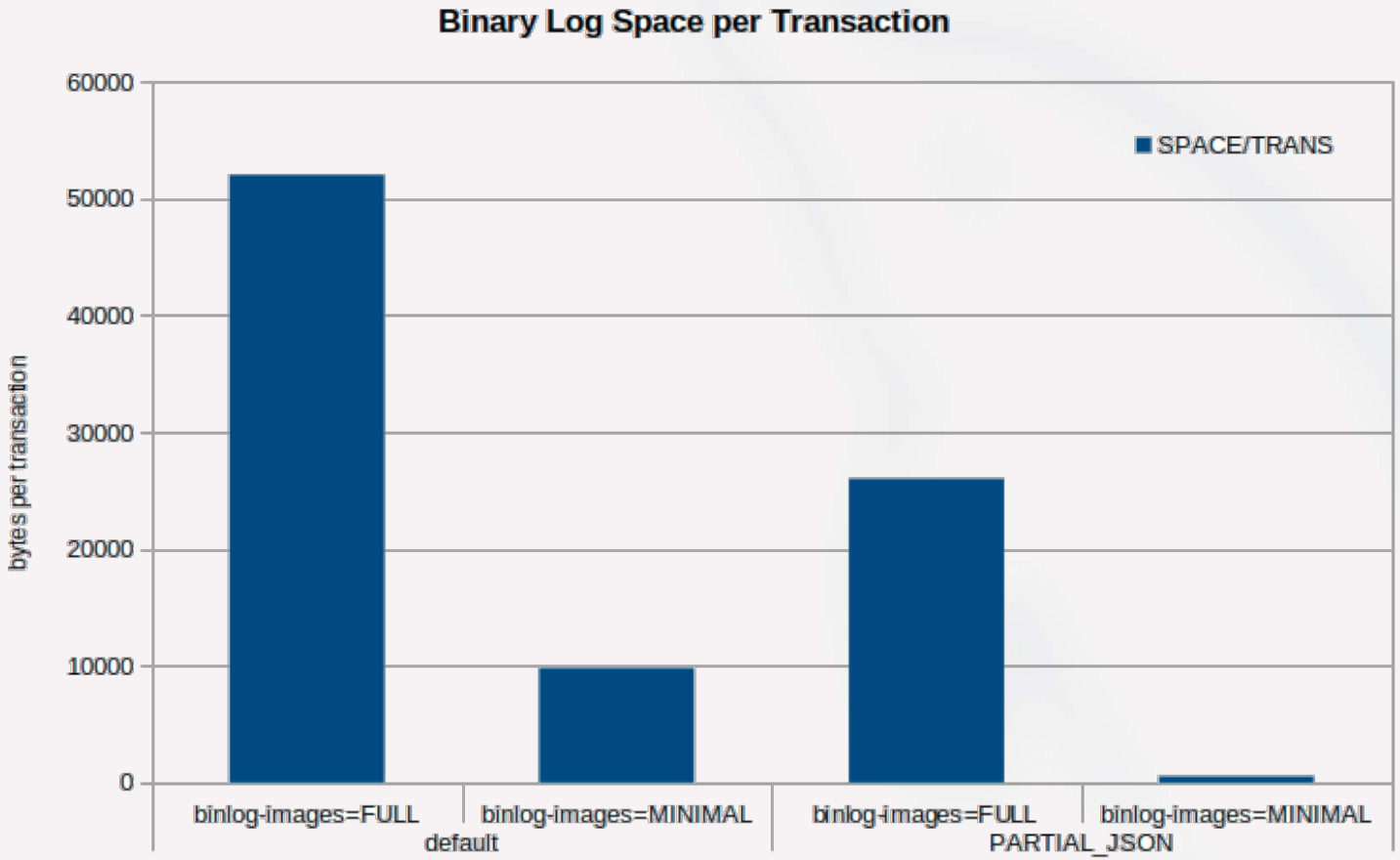
*It's possible to reduce the size of a binary log event (using* `ROW` *based replication). Usually in MySQL row-based replication, each row change event contains two images, a "before" image whose columns are matched against when searching for the row to be updated, and an "after" image containing the changes.*

*It's possible to reduce that to save memory, disk space and network usage. If you don't need the both full images, I recommend to use* `minimal` *row image.*

*I also recommend to replicate only the changes when using JSON columns.*

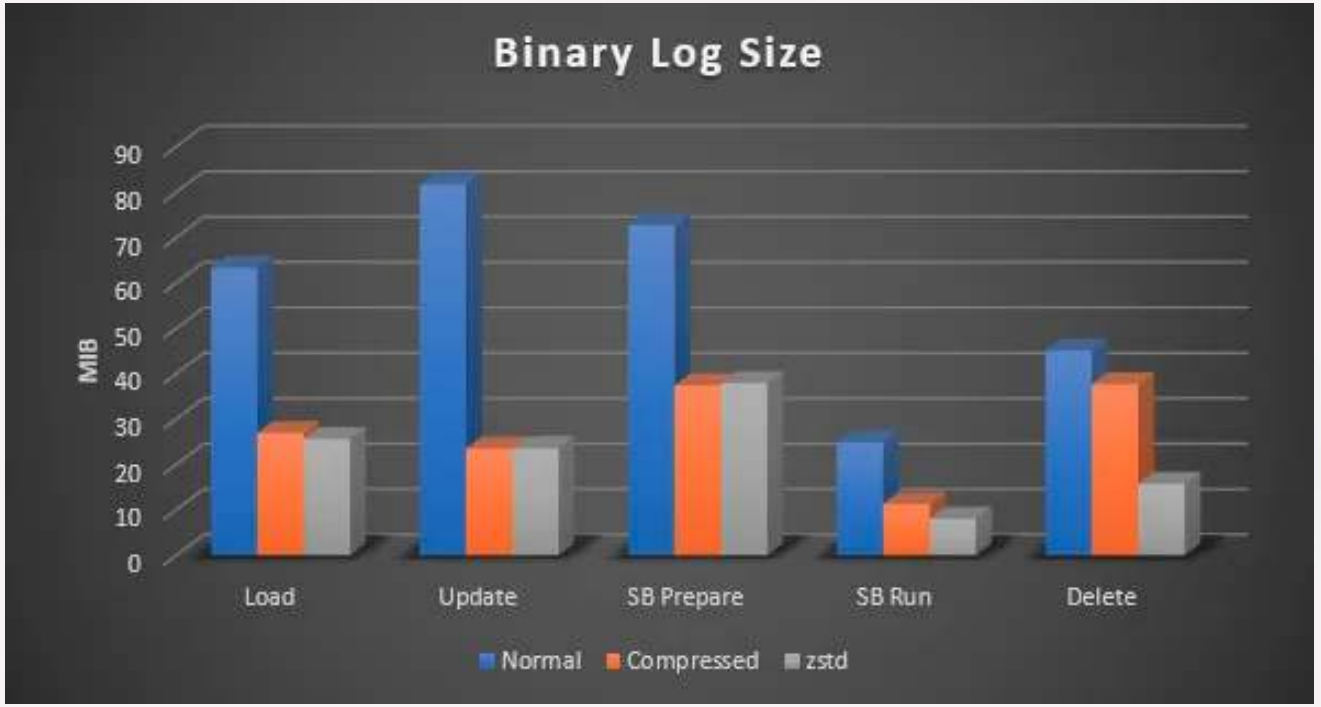
```
binlog_row_image = minimal
binlog_row_value_options = PARTIAL_JSON
binlog_transaction_compression = ON
```

# Binlogs (2)



# Binlogs (3)

*Since MySQL 8.0 it's also possible to compress transaction payloads and then written them to the binary log file as a single event.*




```
binlog_transaction_compression = ON
```

*credits: @JWKrogh - https://mysql.wisborg.dk/2020/05/07/mysql-compressed-binary-logs/*

# Recap: binlogs settings - TIP 19

*You can reduce the size of the binary logs and improve the performance of asynchronous replication by using:*

- `binlog_row_image = minimal`
- `binlog_row_value_options = PARTIAL_JSON`
- `binlog_transaction_compression = ON`

# IO Bound

*disks are slow*

# Are we IO bound ?

*Most of the time people are not aware of having IOPS issues and blame the database.*

*Underestimating or overestimating the IOPS capabilities can be dramatic for MySQL performance.*

# Are we IO bound ?

*Most of the time people are not aware of having IOPS issues and blame the database.*

*Underestimating or overestimating the IOPS capabilities can be dramatic for MySQL performance.*

*How can we detect that we are having IOPS issues ?*

# Are we IO bound ?

*Most of the time people are not aware of having IOPS issues and blame the database.*

*Underestimating or overestimating the IOPS capabilities can be dramatic for MySQL performance.*

*How can we detect that we are having IOPS issues ?*

*On Linux, we have multiple tools that are able to help us in this particular task:*

- `vmstat`
- `iostat`
- ...

# vmstat

```
[fred@imac ~] $ vmstat 1 10
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  5 365568 415996 250172 6762500    1    1     7    38   14   24 16 12 71  0  0
 0  2 365568 362368 249704 6779348    0    0   336 57476 18294 43754  6 10 41 43  0
 0  3 365568 311696 248496 6712712    0    0    52 216584 21016 50609  5 18 33 44  0
 2  3 365568 424408 248500 6712712    0    0     4     0 95959 198530  6 20 27 47  0
 2  2 365568 477200 248508 6715224    0    0   472 12916 28161 63569  7 11 12 70  0
 0  4 365568 304104 248560 6715848    0    0   548 249160 17465 43146  9 17 26 48  0
 0  3 365568 389236 248572 6716348    0    0   512 38140 15197 37865  7  9 31 54  0
```

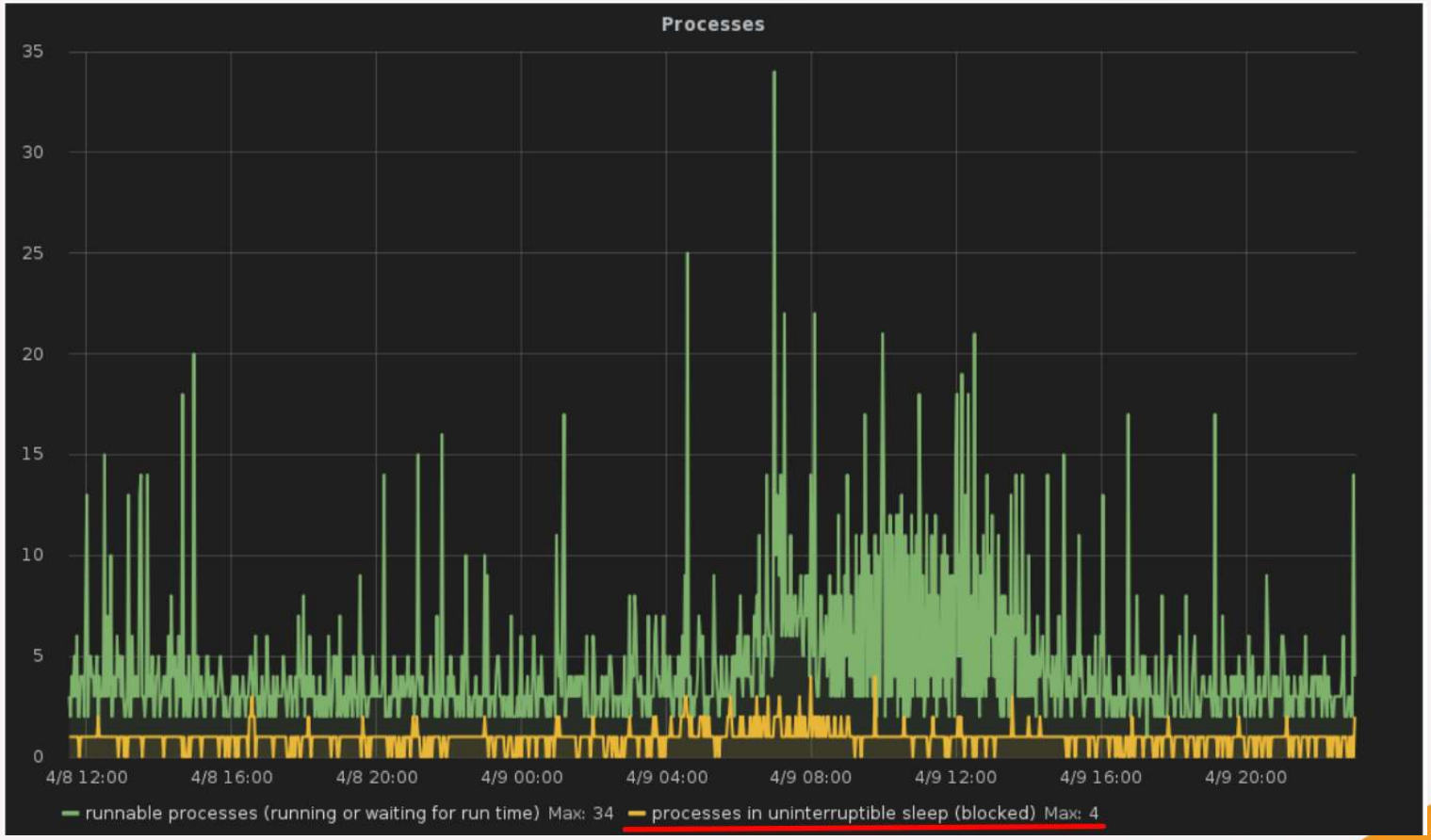
# vmstat

```
[fred@imac ~] $ vmstat 1 10
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  5 365568 415996 250172 6762500    1    1     7    38   14   24 16 12 71  0  0
 0  2 365568 362368 249704 6779348    0    0   336 57476 18294 43754  6 10 41 43  0
 0  3 365568 311696 248496 6712712    0    0    52 216584 21016 50609  5 18 33 44  0
 2  3 365568 424408 248500 6712712    0    0     4     0 95959 198530  6 20 27 47  0
 2  2 365568 477200 248508 6715224    0    0   472 12916 28161 63569  7 11 12 70  0
 0  4 365568 304104 248560 6715848    0    0   548 249160 17465 43146  9 17 26 48  0
 0  3 365568 389236 248572 6716348    0    0   512 38140 15197 37865  7  9 31 54  0
```

*The* **b** *column illustrates the number of processes blocked waiting for I/O to complete. This is often a sign of IO problem of course.*

# vmstat

*If we illustrate that info, we can get a comprehensible graph like this one:*



# iostat

```
[fred@imac ~] $ sudo iostat -dx 2 sda
Linux 5.12.9-300.fc34.x86_64 (imac)     15/06/21        _x86_64_        (4 CPU)

Device            r/s     rkB/s   rrqm/s  %rrqm r_await rareq-sz     w/s     wkB/s   wrqm/s  %wrqm w_await wareq-sz     d/s     dkB/s   drqm/s  %drqm d_await dareq-sz     f/s f_await  aqu-sz   %util
sda              1.07     22.82     0.32  23.25   17.47    21.42    6.73    154.40     3.42  33.69   18.31    22.94    0.00      0.00     0.00   0.00    0.00     0.00    0.79   19.09    0.16    2.28

Device            r/s     rkB/s   rrqm/s  %rrqm r_await rareq-sz     w/s     wkB/s   wrqm/s  %wrqm w_await wareq-sz     d/s     dkB/s   drqm/s  %drqm d_await dareq-sz     f/s f_await  aqu-sz   %util
sda              0.50     32.00     0.00   0.00   80.00    64.00  105.00    1056.00    15.50  12.86   17.82    10.06    0.00      0.00     0.00   0.00    0.00     0.00   26.50   36.92    2.89   99.35

Device            r/s     rkB/s   rrqm/s  %rrqm r_await rareq-sz     w/s     wkB/s   wrqm/s  %wrqm w_await wareq-sz     d/s     dkB/s   drqm/s  %drqm d_await dareq-sz     f/s f_await  aqu-sz   %util
sda              0.00      0.00     0.00   0.00    0.00     0.00   96.00    1322.00    23.00  19.33   19.79    13.77    0.00      0.00     0.00   0.00    0.00     0.00   24.00   39.77    2.85   95.15

Device            r/s     rkB/s   rrqm/s  %rrqm r_await rareq-sz     w/s     wkB/s   wrqm/s  %wrqm w_await wareq-sz     d/s     dkB/s   drqm/s  %drqm d_await dareq-sz     f/s f_await  aqu-sz   %util
sda              8.50    208.00    40.50  82.65   37.41    24.47   97.50    1480.00    18.00  15.58   20.17    15.18    0.00      0.00     0.00   0.00    0.00     0.00   25.00   38.80    3.25   97.05

Device            r/s     rkB/s   rrqm/s  %rrqm r_await rareq-sz     w/s     wkB/s   wrqm/s  %wrqm w_await wareq-sz     d/s     dkB/s   drqm/s  %drqm d_await dareq-sz     f/s f_await  aqu-sz   %util
sda              1.00      4.00     0.00   0.00   85.50     4.00   71.50    7852.00    41.00  36.44   25.93   109.82    0.00      0.00     0.00   0.00    0.00     0.00   16.50   35.64    2.53   58.75
```

*We need to check the column* **`%util`***: the percentage of elapsed time during which I/O requests were issued to the device. On devices serving requests in parallel, such as RAID arrays and modern SSDs, when this number is near 100%, that means the disk is saturated.*

# Performance_Schema

`Performance_Schema` *can also provide some information about the time MySQL is waiting on disk for some operations:*

```
mysql> SELECT source, event_name, format_pico_time(TIMER_END-TIMER_START) wait
       FROM performance_schema.events_waits_current
       WHERE event_name like'%innodb%file';
+-----------------+---------------------------------------+----------+
| source          | event_name                            | wait     |
+-----------------+---------------------------------------+----------+
| fil0fil.cc:8427 | wait/io/file/innodb/innodb_data_file | 2.19 ms  |
| fil0fil.cc:8284 | wait/io/file/innodb/innodb_log_file  | 22.70 ms |
| fil0fil.cc:7731 | wait/io/file/innodb/innodb_log_file  | 20.53 ms |
+-----------------+---------------------------------------+----------+
3 rows in set (0.00 sec)
```

# Knowing the limits

*To know the limit, or to confirm limits we got from the vendor, I usually benchmark the IOPS capabilities of my system. I really encourage you to do the same.*

*If you create a mount point from a RAID, a SSD etc... to become the datadir of your MySQL instance, before installing MySQL, I recommend the use of* **sysbench** *(https://github.com/akopytov/sysbench) and perform a test that will be as close as possible as a MySQL InnoDB workload and the size we expect to have.*

```
$ time sysbench fileio --file-test-mode=rndwr --file-total-size=50G --file-num=10 \
                       --file-extra-flags=direct --threads=8 \
                       --file-io-mode=async --file-fsync-all \
                       --file-block-size=16384 prepare
```

# Knowing the limits (2)

*Get he max IOPS for random write operations:*

```
$ sysbench fileio --file-test-mode=rndwr --file-total-size=50G --file-num=10 \
                  --file-extra-flags=direct --threads=8 \
                  --file-io-mode=async --file-fsync-all \
                  --file-block-size=16384 --report-interval=1 \
                  --events=1000000 run
```

# Knowing the limits (2)

*Get he max IOPS for random write operations:*

```
$ sysbench fileio --file-test-mode=rndwr --file-total-size=50G --file-num=10 \
                  --file-extra-flags=direct --threads=8 \
                  --file-io-mode=async --file-fsync-all \
                  --file-block-size=16384 --report-interval=1 \
                  --events=1000000 run
```

*You can test **`vmstat`** and **`iostat`** at the same time...*

# Knowing the limits (2)

*Get he max IOPS for random write operations:*

```
$ sysbench fileio --file-test-mode=rndwr --file-total-size=50G --file-num=10 \
                  --file-extra-flags=direct --threads=8 \
                  --file-io-mode=async --file-fsync-all \
                  --file-block-size=16384 --report-interval=1 \
                  --events=1000000 run
```

*You can test **`vmstat`** and **`iostat`** at the same time...*

```
File operations:
    reads/s:                      0.00
    writes/s:                     50.13
    fsyncs/s:                     50.13
```

# Recap: is my system IO bound? - TIP 20

*Be aware of the IOPS capabilities of your system and check if you are IO bound.*

- *use* `vmstat` *to check the number of processes blocked waiting for I/O*
- *use* `iostat` *to check the percentage of elapsed time during which I/O requests were issued to the device*
- *use* `Performance_Schema` *to check the time MySQL is waiting on disk for some operations*
- *benchmark the IOPS capabilities of your system using* `sysbench`

Share your ❤️ to MySQL

#mysql #MySQLCommunity

slack

Join our slack channel!

bit.ly/mysql-slack

MySQL

# Questions ?