



MySQL for E-Commerce

Thomas Bley

May 2023

About me

- Senior PHP Developer
- Linux, PHP, MySQL since 3.23
- studied at TU München



Three things are important in the database world:
performance, performance and performance

source: sigmod.org/publications/interviews/pdf/p71-column-winslet.pdf

Schema design

- schema probably root cause of “my DB doesn’t scale”
- use smaller data types (don’t store numbers as varchar)
- use decimal instead of float
- avoid UUIDs as primary keys (use `uuid_short()` or ULIDs)
- use `tinyint` for bools, enums for fixed-value strings
- select only required columns (avoid `select *`)
- avoid queries without indexes (full table scan)
- reduce tables and joins with JSON columns
- use `utf8mb4` or `ascii` as character set for varchar
- use default values instead of nullable columns
- don’t use entity-attribute-value model

→ **write less data, read less data**

→ **read less rows** (less random IO, less pool buffer lookups)

sources: dev.mysql.com/doc/refman/8.0/en/storage-requirements.html

percona.com/blog/2019/11/22/uuids-are-popular-but-bad-for-performance-lets-discuss/

honeybadger.io/blog/uuids-and-ulids/

static.sched.com/hosted_files/perconaliveonline2020/ab/Boosting%20MySQL%20Performance.pdf

vettabase.com/blog/what-does-null-mean-in-sql/

percona.com/blog/queries-for-finding-poorly-designed-mysql-schemas-and-how-to-fix-them/

Type	Storage (Bytes)
TINYINT	1
SMALLINT	2
MEDIUMINT	3
INT	4
BIGINT	8

Data Type	
YEAR	1 byte
DATE	3 bytes
TIME	3 bytes
DATETIME	5 bytes
TIMESTAMP	4 bytes

Advantages of JSON columns

- less tables, less indexes, less joins
- application code can be written quicker and easier:
 - insert, update, delete with 1 query on 1 table on 1 row (no transaction needed)
 - select only on 1 table
- compared to entity-attribute-value model:
 - less complexity, more consistency, much more performance
- table row count grows linearly instead of exponentially
- JSON elements can be queried, updated or indexed directly within SQL
- JSON data can be created or aggregated on relational data
- attributes can be dynamically added to or removed from JSON columns
- elements in JSON arrays can be pre-sorted
- JSON arrays can be capped to a maximum number of elements
- easy to convert PHP objects and arrays to JSON strings

→ JSON columns are best to combine relational and non-relational worlds

sources: dev.mysql.com/doc/refman/8.0/en/json.html
dev.mysql.com/doc/refman/8.0/en/json-function-reference.html



Find the best indexes

- reduce number of indexes, reduce number of columns per index
- use smaller data types for index columns (reduce memory usage)
- ideal: all lookups use indexes or primary keys
- analyze, explain, explain, explain, test, test, test
- keep an eye on slow queries (optimizer sometimes chooses wrong indexes, use force index)
- test new mysql versions (performance, deprecations, changes in optimizer)
- having the right index vs. using it correctly:

```
select count(*) from cart_item where created_at > '2022-01-01';
```

```
1 row in set (0.127 sec)
```

```
select count(*) from cart_item where year(created_at) = '2022';
```

```
1 row in set (2.196 sec)
```

```
explain select count(*) from cart_item where created_at >= '2022-01-01';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	ROWS	filtered	Extra
1	SIMPLE	cart_item	NULL	range	created_at	created_at	4	NULL	2993487	100.00	Using where; Using index

```
explain select count(*) from cart_item where year(created_at) = '2022';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	ROWS	filtered	Extra
1	SIMPLE	cart_item	NULL	index	NULL	created_at	4	NULL	4987011	100.00	Using where; Using index

```
select count(*) from cart_item;
```

```
5074069
```

theory: db.in.tum.de/teaching/ws1819/queryopt/slides/chapter3.pdf

index-merge: percona.com/blog/2009/09/19/multi-column-indexes-vs-index-merge/

Max() vs. order by

- similar queries producing the same result can perform differently
- the optimizer can produce unexpected query plans
- with growing tables, query plans can change
- use index hints when necessary

```
explain select max(pkey) from t1 where some_col < 10000;
```

id	type	possible_keys	key	rows	filtered	Extra
1	range	some_col	some_col	1437473	100.00	Using where; Using index

```
select max(pkey) from t1 where some_col < 10000;
```

```
1 row in set (0.38 sec)
```

```
explain select pkey from t1 where some_col < 10000 order by pkey desc limit 1;
```

id	type	possible_keys	key	rows	filtered	Extra
1	index	some_col	PRIMARY	2	50.00	Using where; Backward index scan

```
select pkey from t1 where some_col < 10000 order by pkey desc limit 1;
```

```
1 row in set (1.11 sec)
```

```
select pkey from t1 use index (some_col) where some_col < 10000 order by pkey desc limit 1;
```

```
1 row in set (0.37 sec)
```

Named locks: GET_LOCK(name, timeout_seconds)

- obtains an exclusive global lock with a given name
 - returns 1 on success
 - returns 0 if attempt timed out
 - returns null on error
- can make parallel requests or scripts execute sequentially
- is managed by current MySQL session, independant from transactions
- released on RELEASE_LOCK(name) or disconnect (e.g. if PHP process stops)
- more efficient than row level locking or transactions, avoids deadlocks
- examples:
 - limit a cron job to one instance running at the same time:
`SELECT GET_LOCK("cron_<cron-job-name>", 1);`
 - process parallel cart requests sequentially:
`SELECT GET_LOCK("cart_<customer-id>", 10);`
 - limit customers to place exactly one order at the same time:
`SELECT GET_LOCK("order_<customer-id>", 10);`

source: dev.mysql.com/doc/refman/8.0/en/locking-functions.html#function_get-lock



Connection handling

- maximum number of open connections (sessions) is often the limiting factor for a database server
- open connection late (ideally on the first query)
- close connection early (esp. when waiting for another external service)
- connect with compression to reduce network traffic
- can save additional hardware for mysql proxies
- persistent connections:
 - every PHP process uses its own mysqli pool (php.net/manual/en/mysqli.quickstart.connections.php)
 - can be buggy (e.g. bugs.php.net/bug.php?id=64549)
- Example:

`bootstrap.php`

```
$db = new PDO('mysql:...;charset=utf8mb4;', ..., [PDO::ATTR_TIMEOUT => 5]);
```

vs.

```
class Repository {
```

```
...
```

```
public function getTasks(Customer $customer): array {  
    $db = $this->app->getDatabase($customer);  
    $statement = $db->query('SELECT id, name FROM tasks ...');  
    return $statement->fetchAll(PDO::FETCH_CLASS, Tasks::class);  
}
```



Cart updates

- most traffic in a shop goes to product data and cart
- reduce number of columns in cart table
 - e.g. customer_id, sku, quantity with primary key on customer_id, sku
- insert / update cart items with 1 query:

```
INSERT INTO cart_item  
  SET customer_id = ?, sku = ?, quantity = ? AS new  
  ON DUPLICATE KEY UPDATE quantity = new.quantity;
```

```
REPLACE INTO cart_item SET customer_id = ?, sku = ?, quantity = ?;
```

- define maximum number of items allowed in a single cart
 - too many items in the cart slow down browser / apps and increase cart calculation time
- define maximum quantity per sku if logistics requires it
 - e.g. max. 5 crates of water per order

source: dev.mysql.com/doc/refman/8.0/en/insert-on-duplicate.html



Limit resultsets: SELECT ... LIMIT

- limit gives quicker response (fetch less data)
- less memory usage for processing less rows
- less data to process in frontend / apps
- page by primary key instead of using OFFSET
 - `SELECT name FROM wishlist LIMIT 100 OFFSET 10000;`

VS.

- `SELECT name FROM wishlist WHERE id > 10000 LIMIT 100;`

→ read less rows, process less rows

source: shopify.engineering/pagination-relative-cursors<https://shopify.engineering/pagination-relative-cursors>

Verify status transitions

- `UPDATE orders SET status = 'processing'`
`WHERE order_id = 42;`

VS

- `UPDATE orders SET status = 'processing', updated_at = now()`
`WHERE order_id = 42 AND status = 'new';`
- verify that dataset was changed in the database:
`if ($db->query($query)->rowCount() !== 1) {...}`



Analytical queries: SELECT ...

- use **read-only replica** whenever possible (reduce traffic on master)
 - use **pre-aggregation** of data to read less data
 - transaction-isolation = READ COMMITTED
 - REPEATABLE-READ: locks acquired during transaction are held for duration of transaction
 - **READ-COMMITTED**: locks that did not match the scan are released after STATEMENT completes, in READ COMMITTED gap locks are never created
 - use a column store database for ad-hoc queries (clickhouse, duckdb, etc.)
 - store data by columns instead of rows → better compression, quicker filtering, aggregation, sorting, etc.
 - vectorized joins, multi-core parallel query execution, S3 storage layer
 - faster sorting (<https://duckdb.org/2021/08/27/external-sorting.html>)
 - materialized views (<https://altinity.com/blog/clickhouse-materialized-views-illuminated-part-1>)
 - requires loading data into column store database
- gives best performance for analytical queries

sources:

percona.com/blog/2012/08/28/differences-between-read-committed-and-repeatable-read-transaction-isolation-levels/

percona.com/blog/2015/01/14/mysql-performance-implications-of-innodb-isolation-modes/

Read data consistently from replicas

- data is replicated asynchronously to replicas
- replica delay can be monitored with: `show replica status; // Seconds_Behind_Source`
- parallel replication enabled by default with MySQL 8 (before single thread)
- request #1:

- update data on master

```
-- connect to master
```

```
INSERT INTO tasks SET id = 42, name = 'foo';
```

```
-- get latest gtid
```

```
SELECT @@GLOBAL.gtid_executed;
```

- store gtid in cookie

- request #2:

- wait for transaction to be applied on replica

```
-- connect to replica
```

```
SELECT WAIT_FOR_EXECUTED_GTID_SET(gtid_from_cookie, 5); -- 0 = success
```

- read data from replica:

```
SELECT name FROM tasks WHERE id = 42;
```

source: youtube.com/watch?v=9caro2QNcww



Customer data search: low traffic search

MySQL fulltext index

- can be implemented quickly and easily
- can be fast enough for low traffic search
- no extra hardware for search functionality needed
- stays consistent (with index update on transaction commit)

```
CREATE TABLE customer_search (  
    customer_id int UNSIGNED NOT NULL PRIMARY KEY,  
    search varchar(1024) NOT NULL, -- e.g. "firstname lastname address"  
    FULLTEXT (search)  
);
```

```
SELECT customer.* FROM customer, customer_search  
WHERE customer.id = customer_search.customer_id  
    AND MATCH(customer_search.search) AGAINST (? in boolean mode)  
LIMIT 100;
```

- query example: "Frank-Walther" Wattstr* +Meier -Franz
- query input requires sanitizing for "-+<>~*@"
 e.g. "++foo", "**foo" or "foo+" not allowed

source: dev.mysql.com/doc/refman/8.0/en/fulltext-boolean.html



Product data search: high traffic search

- most traffic in a shop goes to product data and cart
- store each product in a JSON document
- use (in-memory) document store as data source
- example products optimized for fulltext search:
RedisSearch, meilisearch, typesense, Inx-search, Elasticsearch, Algolia
- remember to check licenses and support options
- make vs. buy



Queues

MySQL can be used as a queue:

- for small messages with low volume
- provides transactional safety, example:

```
BEGIN;
```

```
INSERT INTO customer SET id = 42;
```

```
REPLACE INTO queue_crm SET customer_id = 42, num_tries = 0;
```

```
COMMIT;
```

- can process messages in order or out of order, example for **single** consumer:

```
SELECT GET_LOCK('queue_crm', 1);
```

```
SELECT customer_id FROM queue_crm WHERE num_tries < 10 ORDER BY created_at ASC LIMIT 10;
```

```
UPDATE queue_crm SET num_tries = num_tries + 1, last_try = now() WHERE customer_id = 42;
```

```
-- process entry
```

```
DELETE FROM queue_crm WHERE customer_id = 42;
```

```
SELECT RELEASE_LOCK('queue_crm');
```

- releases row locks on RELEASE_LOCK() or disconnect (e.g. if PHP process stops)
- makes it easy to re-process failed messages, example:

```
UPDATE queue_crm SET num_tries = 0 WHERE num_tries = 10;
```

Queues #2

MySQL can be used as a queue:

- can process messages in order or out of order, example for **multiple** consumers:

BEGIN;

```
SELECT customer_id FROM queue_crm WHERE num_tries < 10
ORDER BY created_at ASC LIMIT 10 FOR UPDATE SKIP LOCKED;
-- process entry
-- on success
DELETE FROM queue_crm WHERE customer_id = 42;
-- on failure
UPDATE queue_crm SET num_tries = num_tries + 1, last_try = now() WHERE customer_id = 42;
```

COMMIT;

- releases row locks on COMMIT or disconnect (e.g. if PHP process stops)
- makes it easy to do queue monitoring:

```
SELECT count(*) FROM queue_crm WHERE num_tries >= 10;
```

source: dev.mysql.com/blog-archive/mysql-8-0-1-using-skip-locked-and-nowait-to-handle-hot-rows/



Log data

- use page compression to reduce read and write IO
- using partitions offers quick deletion by ALTER TABLE log TRUNCATE PARTITION ...
- use JSON columns to store structured logs
- store logs outside of database whenever possible (e.g. datadog, elasticsearch)

- ```
CREATE TABLE log (
 customer_id int unsigned not null,
 event json not null,
 created_at datetime not null,
 index(customer_id, created_at)
) COMPRESSION='lz4' PARTITION BY hash(month(created_at)) PARTITIONS 12;
```

- check compression ratio and partition consistency:

```
select name, file_size, allocated_size from information_schema.INNODB_TABLESPACES where name like 'tasks/log%';
```

```
+-----+-----+-----+
| name | file_size | allocated_size |
+-----+-----+-----+
| tasks/log#p#p0 | 1086324736 | 551456768 |
| tasks/log#p#p1 | 1203765248 | 612634624 |
...
```

...

```
check table log;
```

source: [dev.mysql.com/doc/refman/8.0/en/innodb-page-compression.html](https://dev.mysql.com/doc/refman/8.0/en/innodb-page-compression.html)

# Data migration (EAV → JSON)

Using SQL to copy data from old schema to new schema (old table(s) to new table):

```
INSERT INTO customers.customer (email, attributes, created)
SELECT * FROM magento.customer_entity
LEFT JOIN magento.customer_entity_varchar as firstname on
 firstname.entity_id = customer_entity.entity_id and firstname.attribute_id = 5
LEFT JOIN magento.customer_entity_varchar as lastname on
 lastname.entity_id = customer_entity.entity_id and lastname.attribute_id = 7
LEFT JOIN magento.customer_entity_datetime as birthdate on
 birthdate.entity_id = customer_entity.entity_id and birthdate.attribute_id = 11
SET email = customer_entity.email,
 attributes = json_object(
 'first', firstname.value,
 'last', lastname.value,
 'birth', date(birthdate.value),
 'orders', (select count(*) from magento.sales_flat_order
 where customer_id = customer_entity.entity_id)
),
 created = customer_entity.created_at;
```



# Data anonymization

- Use a separate database and views to provide anonymized production data
- Grant permissions to human users only on anonymized data

```
SELECT * FROM customers.customer WHERE id = 1234;
```

```
id: 1234
```

```
email: foo.bar@baz.com
```

```
attributes: {"first": "Thomas", "last": "Bley", "birth": "1930-02-01", "orders": 42}
```

```
created: 2021-05-06 12:41:11
```

```
CREATE or REPLACE VIEW customers_anonymized.customer AS
```

```
SELECT id,
```

```
concat('invalid_', id, '@bringmeister.de') AS email, -- invalid_1234@bringmeister.de
```

```
json_object(
```

```
 'first', concat('first_', id), -- first_1234
```

```
 'last', concat('last_', id), -- last_1234
```

```
 'birth', '1980-01-02',
```

```
 'orders', attributes->>"$.orders" -- 1980-01-02, 42
```

```
) AS attributes,
```

```
created
```

```
FROM customers.customer;
```



# Pseudo random order ID generation

Order IDs should be increasing, but increasing with some “randomness”:

- generate Order ID before saving the order
- increment Order ID sequence with random values
- `LAST_INSERT_ID()`: returns automatically generated value
- `RAND()`: Returns a random float value in the range  $0 \leq \text{value} < 1.0$
- can be an alternative to UUID / ULID

```
CREATE TABLE order_sequence (order_id int unsigned not null);
INSERT INTO order_sequence SET order_id = 1; // start with Order ID 1

-- update order_id = order_id + rand(5, 10) returning order_id
UPDATE order_sequence SET order_id = LAST_INSERT_ID(order_id + FLOOR((RAND() * 6) + 5));
SELECT LAST_INSERT_ID();
```

→ generates an randomly increasing Order ID sequence

e.g. 6, 16, 25, 39, 46, 56, 65, 73, 78, 85, 90, ...

source: [dev.mysql.com/doc/refman/8.0/en/information-functions.html#function\\_last-insert-id](https://dev.mysql.com/doc/refman/8.0/en/information-functions.html#function_last-insert-id)



# Foreign keys, triggers and procedures

## skip foreign keys

- require additional indexes
- require additional locking and lookups on related tables for insert, update or delete, can cause deadlocks, slow for data imports
- cascading logic might be different from required business logic
- not supported by some tools (e.g. gh-ost), only work inside a single db server
- can have bugs ([itw01.com/8Q76OEZ.html](http://itw01.com/8Q76OEZ.html))
- locked data types, cascaded write not in binary log ([code.openark.org/blog/mysql/things-that-dont-work-well-with-mysqls-foreign-key-implementation](http://code.openark.org/blog/mysql/things-that-dont-work-well-with-mysqls-foreign-key-implementation))
- only working inside a single server (not for sharding)

## skip triggers and procedures

- logic is limited to SQL functionality
- cannot connect to external systems (e.g. send an email)
- difficult to update atomically
- often cause unwanted side effects, e.g. cyclic execution

→ better to keep application logic inside application code, use MySQL only as storage

# Legacy code and rewrites

- identify and optimize **20% of slowest queries**
- verify backups
- remove unused indexes (make invisible first), create better indexes
- remove unused columns and tables
- **remove historic data** if possible (GDPR)
- optimize database schema (reduce joins with JSON columns)
- analyze data quality to avoid bugs (esp. historic data)
- use character set utf8mb4
- enable MySQL access log for local development
- use caching when data changes rarely
- fill caches in the background or use random expiries
- **write tests** to verify that data is written correctly to the database
  - start tests with empty tables, create test data on the fly
  - put integrations tests inside a transaction (setUp())
  - rollback changes when tests finish (tearDown())
  - allow tests to run in parallel





# Integration tests

- start tests with **empty** tables
- wrap tests inside a **transaction**
- verify that **data** is written **correctly** to the database
- example:

```
class ExampleRepositoryTest extends TestCase
{
 protected function setUp(): void
 {
 $this->app->getDatabase()->beginTransaction();
 }

 protected function tearDown(): void
 {
 $this->app->getDatabase()->rollBack();
 }

 public function testCreateExample(): void
 {
 // insert into ...
 // select from ...
 // assert ...
 }
}
```



# Mass delete data

Mass delete data can be much quicker by copying data to a second table:

```
DELETE FROM sometable WHERE created_at <= date_sub(now(), INTERVAL 3 MONTH);
```

vs.

```
CREATE TABLE sometable_new LIKE sometable;
```

```
LOCK TABLE sometable write, sometable_new WRITE;
```

```
INSERT INTO sometable_new SELECT * FROM sometable
 WHERE created_at > date_sub(now(), INTERVAL 3 MONTH);
```

```
RENAME TABLE sometable TO sometable_old, sometable_new TO sometable;
```

```
UNLOCK TABLES;
```

```
DROP TABLE sometable_old;
```



# Schema changes

- table writes are blocked by alter table operations
- use tools like gh-ost for background table changes  
(requires double the disk space for a table, does not support foreign keys)
- example:

```
optimize table customers.customer;
```

```
-- can be written as
```

```
alter table customers.customer engine=innodb;
```

```
-- and becomes
```

```
gh-ost \
```

```
--max-load=Threads_running=25 --critical-load=Threads_running=1000 \
```

```
--host=<host> --port=<port> --user="root" --password="<pw>" \
```

```
--chunk-size=100000 --dml-batch-size 10 \
```

```
--initially-drop-ghost-table --initially-drop-old-table \
```

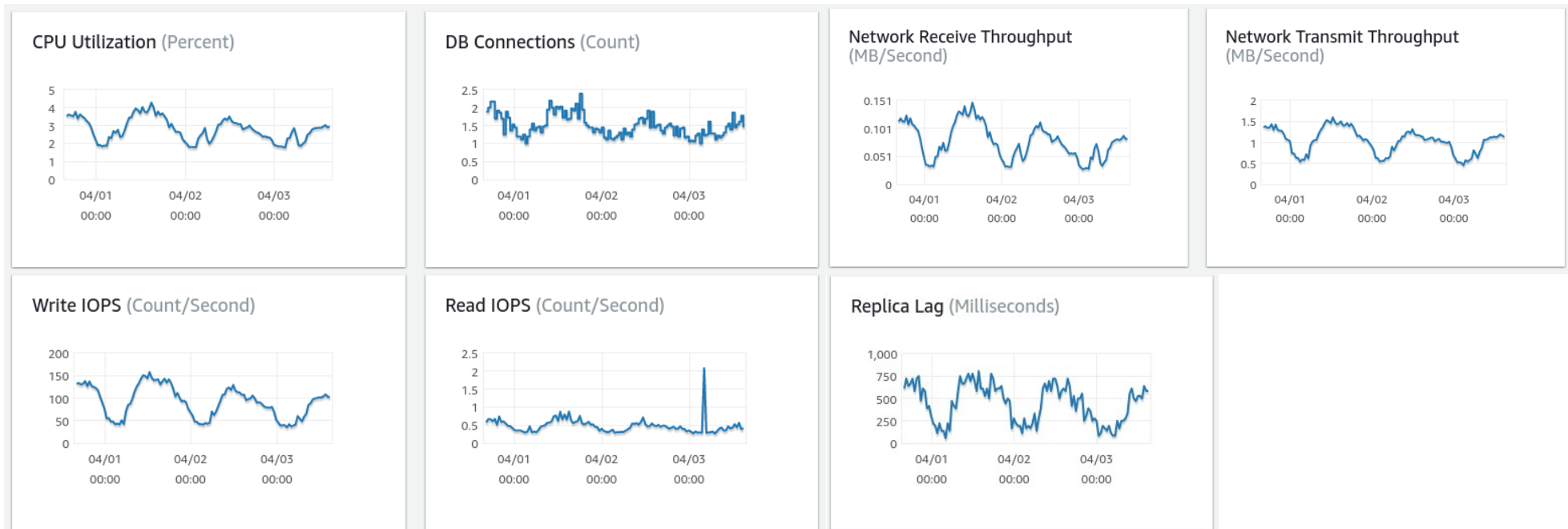
```
--ok-to-drop-table --allow-on-master --execute \
```

```
--database="customers" --table="customer" --alter="engine=innodb"
```

source: [github.com/github/gh-ost](https://github.com/github/gh-ost)

# Monitoring

- log slow queries (server side, slow query log, show processlist)
- log failed queries (application side)
- analyze deadlocks: show engine innodb status;
- monitor replica status: show replica status;  
(Replica\_IO\_Running, Replica\_SQL\_Running, Seconds\_Behind\_Source)
- monitor performance metrics:



sources: [percona.com/blog/2006/07/17/show-innodb-status-walk-through/](http://percona.com/blog/2006/07/17/show-innodb-status-walk-through/)  
[percona.com/blog/2014/10/28/how-to-deal-with-mysql-deadlocks/](http://percona.com/blog/2014/10/28/how-to-deal-with-mysql-deadlocks/)

# Monitor auto\_increment overflows

```
ANALYZE TABLE <table_name>, <table_name>, ...; // update information_schema

SELECT
 t.TABLE_SCHEMA, t.TABLE_NAME, c.COLUMN_NAME, t.AUTO_INCREMENT, c.DATA_TYPE,
 t.AUTO_INCREMENT / (
 CASE DATA_TYPE
 WHEN 'tinyint' THEN IF(COLUMN_TYPE LIKE '%unsigned', 255, 127)
 WHEN 'smallint' THEN IF(COLUMN_TYPE LIKE '%unsigned', 65535, 32767)
 WHEN 'mediumint' THEN IF(COLUMN_TYPE LIKE '%unsigned', 16777215, 8388607)
 WHEN 'int' THEN IF(COLUMN_TYPE LIKE '%unsigned', 4294967295, 2147483647)
 WHEN 'bigint' THEN IF(COLUMN_TYPE LIKE '%unsigned', 18446744073709551615,
 9223372036854775807)
 END / 100) AS used
 FROM information_schema.TABLES t NATURAL JOIN information_schema.COLUMNS c
 WHERE t.AUTO_INCREMENT IS NOT NULL AND c.DATA_TYPE LIKE '%int'
 AND c.COLUMN_KEY = 'PRI' AND t.TABLE_SCHEMA != 'mysql'
HAVING used > 0.8;
```

|  | TABLE_SCHEMA |  | TABLE_NAME |  | COLUMN_NAME |  | AUTO_INCREMENT |  | DATA_TYPE |  | used    |  |
|--|--------------|--|------------|--|-------------|--|----------------|--|-----------|--|---------|--|
|  | orders       |  | order      |  | id          |  | 3435973836     |  | int       |  | 85.0000 |  |

source: [superuser.com/questions/729004/nagios-monitor-mysql-auto-increment-overflow](https://superuser.com/questions/729004/nagios-monitor-mysql-auto-increment-overflow)

# Server configuration

- `innodb_dedicated_server`: on
- `innodb_buffer_pool_size`: 75-80% of system memory  
(best case: active workset fits into pool buffer)
- `innodb_flush_method`: `O_DIRECT_NO_FSYNC` (skip kernel file system cache)
- `innodb_use_fdatasync`: on (skip flushing metadata)
- `innodb_flush_log_at_trx_commit`: 2 (default 1, less durability)
- `sync_binlog`: 0 or 1 (keeps binlog/replicas consistent)
- `innodb_adaptive_hash_index`: off (faster for many concurrent joins)
- `max_heap_table_size` / `tmp_table_size`: 128M (used for joins, sorting, subqueries)
- `gtid-mode`: on, `enforce_gtid_consistency`: on
- `wait_timeout`: 120 (4x `max_execution_time`)
- `event_scheduler`: off (when you don't need it)
- `performance_schema`: off (when you don't need it)

sources: [percona.com/blog/2016/04/12/is-adaptive-hash-index-in-innodb-right-for-my-workload/](http://percona.com/blog/2016/04/12/is-adaptive-hash-index-in-innodb-right-for-my-workload/)  
[dev.mysql.com/doc/refman/8.0/en/internal-temporary-tables.html](http://dev.mysql.com/doc/refman/8.0/en/internal-temporary-tables.html)  
[dev.mysql.com/doc/refman/8.0/en/innodb-parameters.html](http://dev.mysql.com/doc/refman/8.0/en/innodb-parameters.html)  
[percona.com/blog/mysql-zfs-performance-update/](http://percona.com/blog/mysql-zfs-performance-update/)  
[dev.mysql.com/doc/refman/8.0/en/innodb-dedicated-server.html](http://dev.mysql.com/doc/refman/8.0/en/innodb-dedicated-server.html)



# Additional resources

- InnoDB Performance Optimization: [youtube.com/watch?v=qMmSwhSlnX0](https://youtube.com/watch?v=qMmSwhSlnX0)
- Introduction into MySQL Query Tuning: [youtube.com/watch?v=1pxcOgzb6aU](https://youtube.com/watch?v=1pxcOgzb6aU)
- Tuning InnoDB Primary Keys: [percona.com/blog/2018/07/26/tuning-innodb-primary-keys/](https://percona.com/blog/2018/07/26/tuning-innodb-primary-keys/)
- MySQL 8.0 – locking details: [lefred.be/content/mysql-8-0-locking-details/](https://lefred.be/content/mysql-8-0-locking-details/)
- Why joins take so much performance? [de.wikipedia.org/wiki/Joinalgorithmen](https://de.wikipedia.org/wiki/Joinalgorithmen)
- Backup Performance Comparison  
[percona.com/blog/dump-performance-comparison-mysqldump-vs-mysql-shell-utilities-vs-mydumper/](https://percona.com/blog/dump-performance-comparison-mysqldump-vs-mysql-shell-utilities-vs-mydumper/)
- Linux OS Tuning for MySQL Database Performance  
[percona.com/blog/2018/07/03/linux-os-tuning-for-mysql-database-performance/](https://percona.com/blog/2018/07/03/linux-os-tuning-for-mysql-database-performance/)
- Millions of Queries per Second  
[percona.com/blog/2017/01/06/millions-queries-per-second-postgresql-and-mysql-peaceful-battle-at-modern-demanding-workloads/](https://percona.com/blog/2017/01/06/millions-queries-per-second-postgresql-and-mysql-peaceful-battle-at-modern-demanding-workloads/)

# Thanks for listening!

## Questions?

download slides:  
[github.com/thomasbley/talks](https://github.com/thomasbley/talks)

follow me:  
[phpc.social/@thbley](https://phpc.social/@thbley)