


# Showdown: MySQL 8 vs PostgreSQL 10

 [hackernoon.com/showdown-mysql-8-vs-postgresql-10-3fe23be5c19e](https://hackernoon.com/showdown-mysql-8-vs-postgresql-10-3fe23be5c19e)

May 23, 2018

[Kenn Ejima](#)

Founder @ <https://dumper.io> | Full-stack software developer | Scaled mobile backend from zero to 30 million users in 3 years | Previously CTO @ East Meet East



Now that [MySQL 8](#) and [PostgreSQL 10](#) are out, it's a good time to revisit how the two major open source relational databases compete against each other.

Before these versions, the general perception has been that while Postgres is superior in feature sets and its pedigree, MySQL is more battle tested at scale with massive concurrent reads /writes.

But with the latest releases, the gap between the two has gotten significantly narrowed.

## Feature Comparison

Let's take a look at the "trendy" features that we all love to talk about.

Features	MySQL 8	PostgreSQL 10
<b>Query &amp; Analytics</b>		
Common Table Expressions (CTEs)	✓ New	✓
Window Functions	✓ New	✓
<b>Data Types</b>		
JSON support	✓ Improved	✓
GIS / SRS	✓ Improved	✓
Full-text Search	✓	✓
<b>Scalability</b>		
Logical Replication	✓	✓ New
Semi-synchronous Replication	✓	✓ New
Declarative Partitioning	✓	✓ New

It used to be easy to say that MySQL works best for online transactions, and PostgreSQL works best for analytical processes. But not anymore.

Common Table Expressions (CTEs) and window functions have been the main reason to choose PostgreSQL. But now, traversing an `employees` table recursively with a reference to the `boss_id` in the same table, or finding a median value (or 50% percentile) in a sorted result are no longer a problem on MySQL.

Lack of configuration flexibility with replication on PostgreSQL was the reason why Uber switched to MySQL. But now with logical replication, zero downtime upgrade is possible by creating a replica with a newer version of Postgres and switching over to it. Truncating a stale partition in a huge time-series event table is much easier, too.

In terms of features, both databases are now on par with each other.

## Where are the differences?

---

Now, we are left with a question—what are the reasons to pick one over the other, then?

**Ecosystem** is one of those factors. MySQL has a vigorous ecosystem with variants such as MariaDB, Percona, Galera, etc. as well as storage engines other than InnoDB, but that can also be overwhelming and confusing. Postgres has had limited high-end options, but that will change with new features introduced with the latest version.

**Governance** is another factor. Everyone feared when Oracle (or originally, SUN) bought MySQL, that they would ruin the product, but that hasn't been the case so far for the past ten years. In fact, the development accelerated after the acquisition. Postgres has a solid history of working governance and collaborative community.

**Architectural fundamentals** don't change often, and it's worth revisiting as those aren't discussed much in detail these days.

Here's a refresher:



## Process vs Thread

---

As Postgres forks off a child process to establish a connection, it can take up to 10 MB per connection. The memory pressure is bigger compared to MySQL's thread-per-connection model, where the default stack size of a thread is at 256KB on 64-bit platforms. (Of course, thread-local sort buffers, etc. make this overhead less significant, if not negligible, but still.)

Even though copy-on-write saves some of the shared, immutable memory state with the parent process, the basic overhead of being a process-based architecture is taxing when you have 1,000+ concurrent connections, and it can be one of the most important factors for capacity planning.

That is, say if you run a Rails app on 30 servers, where each has 16 CPU cores and 32 Unicorn workers, you have 960 connections. Probably less than 0.1% out of all apps will ever reach beyond that scale, but it's something to keep in mind.

## Clustered Index vs Heap Table

---

A clustered index is a table structure where rows are directly embedded inside the B-tree structure of its primary key. A (non-clustered) heap is a regular table structure filled with data rows separately from indexes.

With a clustered index, when you look up a record by the primary key, a single I/O will retrieve the entire row, whereas non-clustered always require at least two I/Os by following the reference. The impact can be significant as foreign key reference and joins will trigger primary key lookup, which account for vast majority of queries.

A theoretical downside of clustered index is that it requires twice as many tree node traversal when you query with a secondary index, as you first scan over the secondary index, then walk through the clustered index, which is also a tree.

But given a modern convention to have an auto-increment integer as a primary key<sup>1</sup>—it's called a surrogate key—it is almost always desirable to have a clustered index. It is more so if you do a lot of `ORDER BY id` to retrieve the most recent (or oldest) N records, which I believe applies to most.

[1] UUID as a primary key is a terrible idea, by the way—cryptographic randomness is utterly **designed to kill** locality of reference, hence the performance penalty.

Postgres does not support clustered index, whereas MySQL (InnoDB) does not support heap. But either way, the difference should be minor if you have a large amount of memory.

## Page Structure and Compression

---

Both Postgres and MySQL have page-based physical storage. (8KB vs 16KB)

From Introduction to PostgreSQL physical storage

On PostgreSQL, the page structure looks like the image on the left.

It contains some headers which we are not going to cover here, but they hold metadata about the page. Items after the header is an array identifier composed of `(offset, length)` pairs pointing to tuples, or data rows. Keep in mind on Postgres, multiple versions of the same record can be stored in the same page in this manner.

MySQL's tablespace structure is similar to Oracle's in that it has multiple hierarchical layers of segment, extent, page and row.

Also it has a separate segment for UNDO, called "rollback segment." Unlike Postgres, MySQL will keep multiple versions of the same record in a separate area.

A row must fit in a single page on both databases, which means a row must be smaller than 8KB. (At least 2 rows must fit in a page on MySQL, which coincidentally is  $16\text{KB} / 2 = 8\text{KB}$ )

So what happens when you have a large JSON object in a column?

Postgres uses TOAST, a dedicated shadow table storage. The large object is pulled out when and only when the row and column is selected. In other words, a large chunk of black box won't pollute your precious cache memory. It also supports compression on the TOASTed objects.



MySQL has a more sophisticated feature called Transparent Page Compression, thanks to the contribution from a high-end SSD storage vendor, Fusion-io. It is specifically designed to work better with SSDs, where write volume is directly correlated to the device's lifetime.

Compression on MySQL works not only on off-page large objects, but on all pages. It does that by using hole punching in a sparse file, which is supported by modern filesystems such as ext4 or btrfs.

For more details, see: Significant performance boost with new MariaDB page compression on FusionIO

## Overhead in UPDATES

---

Another feature that is often missed, but has a major impact on performance and is potentially the most controversial topic, is UPDATES.

It was another reason why Uber switched away from Postgres, which provoked many Postgres advocates to refute it.

- MySQL Might Be Right for Uber, but Not for You
- A PostgreSQL Response To Uber (PDF)

Both are MVCC databases that keep multiple versions of data for isolation.

To do that, Postgres keeps old data in the heap until VACUUMed, whereas MySQL moves old data to a separate area called rollback segments.

On Postgres, when you try to update, the entire row must be duplicated, as well as an index entry pointing to it. This is partly because Postgres does not support clustered index, the physical location of a row referenced from an index is not abstracted out by a logical key.

To solve this problem, Postgres uses Heap Only Tuples (HOT) to not update index when it's possible. But if updates are frequent enough (or if a tuple is large), the history of tuples can easily flow out of the page size of 8KB, spanning multiple pages and limit the effectiveness of the feature. The timing of pruning and/or defragmenting depends on the heuristics. Also, setting fillfactor less than 100 kills space efficiency—a hard trade-off that you shouldn't worry about at a table creation time.

This limitation goes even deeper; because the index tuples do not have any information about transactions, it had long been impossible to support Index-Only Scans until 9.2. It's one of the oldest, most important optimization methods supported by all major databases including MySQL, Oracle, IBM DB2 and Microsoft SQL Server. But even with the latest version, Postgres can't fully support Index-Only Scans when there are a bunch of UPDATES that set dirty bits in Visibility Map, and often choose Seq Scan when we don't want.

On MySQL, updates occur in-place, and the old row data is stashed in a separate area called rollback segment. The consequence is that you don't need VACUUM, and commits are very fast while rollbacks are relatively slow, which is a preferable trade-off for majority of use cases.

It is also smart enough to purge history as soon as possible. If the isolation level of a transaction is set as **READ-COMMITTED** or lower, the history is purged when the statement is completed.

The size of transaction history does not affect the main page. Fragmentation is a non-issue. Hence the better, more predictable overall performance on MySQL.

## Garbage Collection

---

VACUUM on Postgres is very costly as it works in the main heap area, creating a direct resource contention. It feels exactly like a garbage collection in programming languages—it gets in the way and gives you pause at random.

Configuring autovacuum for a table with billions of records remains a challenge.

Purge on MySQL can also be heavy, but as it runs with dedicated threads inside the separate rollback segment, it does not adversely affect the read concurrency in any way. It's much less likely that bloated rollback segments slow you down, even with the default settings.

A busy table with billions of records does not cause history bloat on MySQL, and things such as file size on storage and query performance are pretty much predictable and stable.

## Logs and Replication

---

Postgres has a single source of truth for transaction history called Write Ahead Log (WAL). It is also used for replication, and the new feature called Logical Replication decodes the binary content into more digestible logical statements on the fly, which allows fine-grained control over the data.

MySQL maintains two separate logs: 1. InnoDB-specific redo logs for crash recovery, and 2. binary log for replication and incremental backup.

Redo logs on InnoDB are, as with Oracle, a maintenance-free circular buffer that won't grow over time, and only created at a fixed size at the boot time. This design guarantees that a sequential, consecutive area is reserved on the physical device, which leads to a higher

performance. Bigger redo logs yield higher performance, at the cost of recovery time from crash.

With the new replication features added to Postgres, I'd call it a tie.

## TL;DR

---

Surprisingly enough, it turns out that the general perception still holds; MySQL works best for online transactions, and PostgreSQL works best for append only, analytical processes such as data warehousing<sup>2</sup>.

[2] When I say Postgres is great for analytics, I mean it. In case you don't know about TimescaleDB, it's a wrapper on top of PostgreSQL that allows you to INSERT 1 million records per second, 100+ billion rows per server. Crazy stuff. No wonder why Amazon chose PostgreSQL as its base for Redshift.

As we saw in this article, vast majority of complications with Postgres stem from its append-only, overly redundant heap architecture.

Future versions of Postgres will probably require a major revamp of its storage engine. You don't have to take my word for it—it's actually discussed on the official wiki, which suggests that it's time to take some good ideas back from InnoDB.

It's been said time and time again that MySQL is playing catch-up with Postgres, but this time, the tide has changed.