PERCONA

Building Your Own PostgreSQL DBAs from Available Materials

Dave Stokes
@Stoker
David.Stokes@Percona.com

Talk Proposal

I *used to* be the Certification Manager for MySQL AB (Sun Microsystems and Oracle) and I would *constantly hear* from hiring managers that *it was hard to find qualified MySQL DBAs* ***but it was impossible to find qualified PostgreSQL DBAs***.

So if we need more PostgreSQL DBAs can we build them, if not from scratch, from MySQL DBAs?

I have been delivering a series on PostgreSQL for MySQLs that has a very good response and it turns out that MySQL DBAs can learn another database easily.

This talk will compare and contrast what MySQL DBAs are used to and how to 'transpose' their knowledge to PG.

So if *you* have need for a well trained DBA that knows PostgreSQL then you may have a resource in the MySQL DBA you already know!

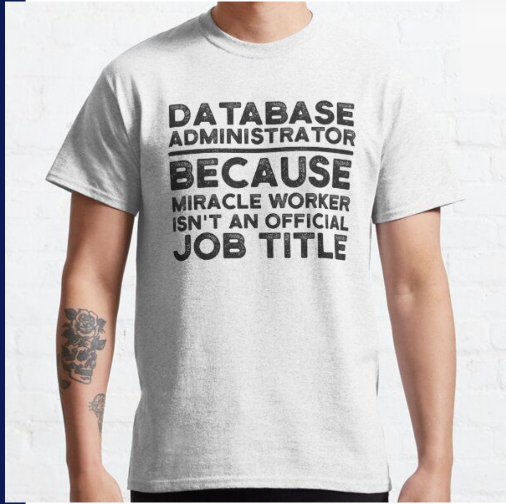
https://www.investopedia.com/

A **make-or-buy decision** is an act of choosing between manufacturing a product in-house or purchasing it from an external supplier.

Also referred to as an outsourcing decision, a make-or-buy decision compares the costs and benefits associated with producing a necessary good or service internally to the costs and benefits involved in hiring an outside supplier for the resources in question.

nize MySQL DBAs?

DATABASE
ADMINISTRATOR
Someone who does precision
guesswork based on unreliable
data provided by those of
questionable knowledge
See also wizard, magician



PostgreSQL versus MySQL differences

Both:

Relational Database Management Systems
Open Source
Popular
Old enough to allowed to drink   (therefore seen as ‘not cool’ by some)

PostgreSQL:

Better SQL Standard Support
Governed by mailing list, consensus
Active community

MySQL:

‘Easier’
Governed (?) by Oracle
Active community

'The devil is in the details'

Ludwig Mies Van Der Rohe.

You found one!

So you find a likely MySQL DBA that you would like to convert. Congratulations!

You might mention that they will have:

- Better skills
- Cross training
- Enhanced job opportunities
- And the ability to now complain knowling about two databases!

So where do you start?

1. **Different approaches to same problems**
2. **New tools**
3. **The basics are still the basics**
   a. Backups/Restore
   b. Account administration
   c. Performance tuning
   d. Query tuning
4. **The really neat new stuff**
   a. Things like two JSON data types, MERGE, Indexes galore, ….
5. **The OMGHDWSHTPI2023* stuff**

*Oh My Goodness How Do We Still Have This Problem In 2023

Start with an installation

Install server

Get it running

Sudo su - postgres

psql

Create a superuser account

DVD rental database load

First steps

Load whichever PG you want and get dvdrental.tar from https://www.postgresqltutorial.com/wp-content/uploads/2019/05/dvdrental.zip

```
$sudo su - postgres
$psql
postgresql=# CREATE DATABASE dvdrental;
postgresql=# exit;
#pgrestore -U postgres -d dvdrental dvdrental.tar
```

(still as user ‘postgres’)

```
$createuser –interactive -s <user>
```

The -s is for superuser

Yup this is dangerous as superuser bypasses some checks but remember you candidate is an experienced DBA (or should be)

Back in the <user> account

```
$psql -d dvdrental
dvdrental=#
```

\d commands

```
dvdrental=# \dt
             List of relations
 Schema |      Name       | Type  |  Owner
--------+-----------------+-------+----------
 public | actor           | table | postgres
 public | address         | table | postgres
 public | category        | table | postgres
 public | city            | table | postgres
 public | country         | table | postgres
 public | customer        | table | postgres
 public | film            | table | postgres
 public | film_actor      | table | postgres
 public | film_category   | table | postgres
 public | inventory       | table | postgres
 public | language        | table | postgres
 public | payment         | table | postgres
 public | rental          | table | postgres
 public | staff           | table | postgres
 public | store           | table | postgres
(15 rows)
```

The Sakila database has been used in the MySQL arena for a very long time in documentation, exams, blogs, and more.

This database is very similar.

There is no `SHOW CREATE TABLE`

```
dvdrental=# show create table actor;

ERROR:  syntax error at or near "create"

LINE 1: show create table actor;

             ^

dvdrental=# \d actor;
```

```
                                                         Table "public.actor"
   Column    |            Type             | Collation | Nullable |                 Default
-------------+-----------------------------+-----------+----------+-----------------------------------------
 actor_id    | integer                     |           | not null | nextval('actor_actor_id_seq'::regclass)
 first_name  | character varying(45)       |           | not null |
 last_name   | character varying(45)       |           | not null |
 last_update | timestamp without time zone |           | not null | now()
Indexes:
    "actor_pkey" PRIMARY KEY, btree (actor_id)
    "idx_actor_last_name" btree (last_name)
Referenced by:
    TABLE "film_actor" CONSTRAINT "film_actor_actor_id_fkey" FOREIGN KEY (actor_id) REFERENCES actor(actor_id) ON
UPDATE CASCADE ON DELETE RESTRICT
Triggers:
    last_updated BEFORE UPDATE ON actor FOR EACH ROW EXECUTE FUNCTION last_updated()
```

Simple queries work as expected

```
dvdrental=# SELECT *
            FROM actor
            ORDER BY last_name, first_name
            LIMIT 10;
 actor_id | first_name | last_name |      last_update
----------+------------+-----------+------------------------
       58 | Christian  | Akroyd    | 2013-05-26 14:47:57.62
      182 | Debbie     | Akroyd    | 2013-05-26 14:47:57.62
       92 | Kirsten    | Akroyd    | 2013-05-26 14:47:57.62
      118 | Cuba       | Allen     | 2013-05-26 14:47:57.62
      145 | Kim        | Allen     | 2013-05-26 14:47:57.62
      194 | Meryl      | Allen     | 2013-05-26 14:47:57.62
       76 | Angelina   | Astaire   | 2013-05-26 14:47:57.62
      112 | Russell    | Bacall    | 2013-05-26 14:47:57.62
      190 | Audrey     | Bailey    | 2013-05-26 14:47:57.62
       67 | Jessica    | Bailey    | 2013-05-26 14:47:57.62
(10 rows)
```

Simple backup

```
$ pg_dump dvdrental > backup.sql
```

- pg_dump is the name of the ‘backup’ program
- dvdrental is name of the database to be backed up
- Dumping the output to file backup.sql

Equivalent to mysqldump

Simple restore

```
$ sudo su - postgres
$ psql
(psql 14.3 (Ubuntu 2:14.3-3-focal))
Type “help” for help.
dvdrental=# CREATE DATABASE newdvd;
dvdrental=# \q
$ ^d
```

```
$ psql -d newdvd –f backup.sql
```

Cheat Sheet

\c dbname Switch connection to a new database

\l List available databases

\dt List available tables

\d table_name Describe a table such as a column, type, modifiers of columns, etc.

\dn List all schemes of the currently connected database

\df List available functions in the current database

\dv List available views in the current database

\du List all users and their assign roles

SELECT version(); Retrieve the current version of PostgreSQL server

\g Execute the last command again

\s Display command history

\s filename Save the command history to a file

\i filename Execute psql commands from a file

\? Know all available psql commands

\h Get help Eg:to get detailed information on ALTER TABLE statement use the \h ALTER TABLE

\e Edit command in your own editor

\a Switch from aligned to non-aligned column output

\H Switch the output to HTML format

\q Exit psql shell

Goodbye AUTO_INCREMENT, Hello SERIAL data type

Small Serial	2 bytes	1 to 32,767
Serial	4 bytes	1 to 2,147,483,647
Big Serial	8 bytes	1 to 9,223,372,036,854,775,807

Yup, MySQL has a SERIAL (`BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE)` but it is a) not widely used, b) will end up creating two indexes if also declared as the PRIMARY KEY.

We start sneaking in sequences!

```
dvdrental=# CREATE SCHEMA test;
CREATE SCHEMA
dvdrental=# \c test
You are now connected to database "test" as user "percona".
test=# CREATE TABLE x (x SERIAL, y CHAR(20), z CHAR(20));
CREATE TABLE
test=# \d x
```

```
                              Table "public.x"
 Column |     Type      | Collation | Nullable |           Default
--------+---------------+-----------+----------+-------------------------------
 x      | integer       |           | not null | nextval('x_x_seq'::regclass)
 y      | character(20) |           |          |
 z      | character(20) |           |          |
```

Demo

```
test=# INSERT INTO X (y,z) VALUES (100,200),(300,450);
INSERT 0 2
test=# SELECT * FROM x;
 x |           y           |           z
---+-----------------------+-----------------------
 1 | 100                   | 200
 2 | 300                   | 450
(2 rows)
```

INSERT replies with the ***oid*** and the ***count***.
The `count` is the number of rows inserted or updated. `oid` is always 0

Values of ‘x’ generated by server

Table & Sequence created by create table

```
test=# \d
           List of relations
 Schema |  Name   |   Type   |  Owner
--------+---------+----------+---------
 public | x       | table    | percona
 public | x_x_seq | sequence | percona
```

Basic Sequences

```
test=# CREATE SEQUENCE order_id START 1001;
CREATE SEQUENCE
test=# SELECT NEXTVAL('order_id');
 nextval
---------
    1001
(1 row)
```

Using nextval()

```
INSERT INTO
    order_details(order_id, item_id, product_name, price)
VALUES
    (100, nextval('order_item_id'), 'DVD Player', 100),
    (100, nextval('order_item_id'), 'Android TV', 550),
    (100, nextval('order_item_id'), 'Speaker', 250);
```

Versus a series

```
test=# create table test1 as (select generate_series(1,100) as id);
SELECT 100
test=# \d test1
               Table "public.test1"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           |          |


test=# select * from test1 limit 5;
 id
----
  1
  2
  3
  4
  5
(5 rows)
```

Fun with *wrapping* sequences

```
test=# create sequence wrap_seq as int minvalue 1 maxvalue 2 CYCLE;
CREATE SEQUENCE
test=# select NEXTVAL('wrap_seq');
 nextval
---------
       1
(1 row)

test=# select NEXTVAL('wrap_seq');
 nextval
---------
       2
(1 row)

test=# select NEXTVAL('wrap_seq');
 nextval
---------
       1
(1 row)

test=# select NEXTVAL('wrap_seq');
 nextval
---------
       2
(1 row)
```

Checking the details on sequences

```
test=# \d order_id;
                          Sequence "public.order_id"
  Type  | Start | Minimum |       Maximum       | Increment | Cycles? |
Cache
--------+-------+---------+---------------------+-----------+---------+---
----
 bigint |  1001 |       1 | 9223372036854775807 |         1 | no      |
1


test=# \d wrap_seq;
                    Sequence "public.wrap_seq"
  Type   | Start | Minimum | Maximum | Increment | Cycles? | Cache
---------+-------+---------+---------+-----------+---------+-------
 integer |     1 |       1 |       2 |         1 | yes     |     1
```

\ds - list sequences

```
dvdrental=#  \ds
                     List of relations
 Schema |            Name            |   Type   |  Owner
--------+----------------------------+----------+----------
 public | actor_actor_id_seq         | sequence | postgres
 public | address_address_id_seq     | sequence | postgres
 public | category_category_id_seq   | sequence | postgres
 public | city_city_id_seq           | sequence | postgres
 public | country_country_id_seq     | sequence | postgres
 public | customer_customer_id_seq   | sequence | postgres
 public | film_film_id_seq           | sequence | postgres
 public | inventory_inventory_id_seq | sequence | postgres
 public | language_language_id_seq   | sequence | postgres
 public | payment_payment_id_seq     | sequence | postgres
 public | rental_rental_id_seq       | sequence | postgres
 public | staff_staff_id_seq         | sequence | postgres
 public | store_store_id_seq         | sequence | postgres
(13 rows)
```

Using Explain

Query tuning can be tough to learn

Explaining EXPLAIN - MySQL edition

```
SQL > EXPLAIN SELECT Name FROM City WHERE District='Texas' ORDER BY Name\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: City
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4188
     filtered: 10
        Extra: Using where; Using filesort
1 row in set, 1 warning (0.0011 sec)
Note (code 1003): /* select#1 */ select `world`.`city`.`Name` AS `Name` from
`world`.`city` where (`world`.`city`.`District` = 'Texas') order by
`world`.`city`.`Name`
```

Test data

```
test=# CREATE TABLE t1 (id SERIAL PRIMARY KEY);
CREATE TABLE
test=# INSERT INTO t1 SELECT GENERATE_SERIES(1,100000);
INSERT 0 100000
test=# CREATE TABLE t2 (id INT NOT NULL);
CREATE TABLE
test=# INSERT INTO t2 SELECT GENERATE_SERIES(1,100000);
INSERT 0 100000
test=#
```

With and without index – Ignore the ANALYZE for now

```
test=# EXPLAIN (ANALYZE) SELECT 1 FROM t2 WHERE ID=101;    #NO Index
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Seq Scan on t2  (cost=0.00..1693.00 rows=1 width=4) (actual time=0.019..5.641 rows=1 loops=1)
   Filter: (id = 101)
   Rows Removed by Filter: 99999
 Planning Time: 0.054 ms
 Execution Time: 5.658 ms
(5 rows)
test=# EXPLAIN (ANALYZE) SELECT 1 FROM t1 WHERE ID=101;    #YES Index
                                                   QUERY PLAN
----------------------------------------------------------------------------------------------------
-------------
 Index Only Scan using t1_pkey on t1  (cost=0.29..4.31 rows=1 width=4) (actual time=0.090..0.091
rows=1 loops=1)
   Index Cond: (id = 101)
   Heap Fetches: 0
 Planning Time: 0.469 ms
 Execution Time: 0.110 ms
```

This is a good comparison of timings

Options in parens new to a MySQL DBA

And no YAML or XML output

Learning to read the output of EXPLAIN

```
dvdrental=# explain SELECT title, first_name, last_name
dvdrental-# FROM film f
dvdrental-# INNER JOIN film_actor fa ON f.film_id=fa.film_id
dvdrental-# INNER JOIN actor a ON fa.actor_id=a.actor_id;
                              QUERY PLAN
---------------------------------------------------------------------------
 Hash Join  (cost=83.00..196.65 rows=5462 width=28)
   Hash Cond: (fa.actor_id = a.actor_id)
   ->  Hash Join  (cost=76.50..175.51 rows=5462 width=17)
         Hash Cond: (fa.film_id = f.film_id)
         ->  Seq Scan on film_actor fa  (cost=0.00..84.62 rows=5462 width=4)
         ->  Hash  (cost=64.00..64.00 rows=1000 width=19)
               ->  Seq Scan on film f  (cost=0.00..64.00 rows=1000 width=19)
   ->  Hash  (cost=4.00..4.00 rows=200 width=17)
         ->  Seq Scan on actor a  (cost=0.00..4.00 rows=200 width=17)
(9 rows)
```

Connections

MySQL has a series of threads

PostgreSQL needs to fork a new process

There are connection poolers available

OMGHDWSHTPI2023

Vacuum

`VACUUM` reclaims storage occupied by dead tuples*.

In normal PostgreSQL operation, tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present until a `VACUUM` is done.

Therefore it's necessary to do `VACUUM` periodically, especially on frequently-updated tables.
-PG Documentation

MySQL uses as difference MVCC approach that automatically takes care of dead tuples and vacuuming will seem very odd to a MySQL DBA

A **tuple** is **PostgreSQL's internal representation of a row in a table**.

Teach VACUUM and AUTOVACUUM

PostgreSQL's VACUUM command has to process each table on a regular basis for several reasons:

- To recover or reuse disk space occupied by updated or deleted rows.
- To update data statistics used by the PostgreSQL query planner.
- To update the visibility map, which speeds up index-only scans.
- To protect against loss of very old data due to transaction ID wraparound or multixact ID wraparound.

```
test=# create table foo (id int, value int);
CREATE TABLE

test=# insert into foo values (1,1);
INSERT 0 1

test=# update foo set value=2 where id =1;
UPDATE 1
test=# update foo set value=3 where id =1;
UPDATE 1
test=# update foo set value=4 where id =1;
UPDATE 1

test=# select relname, n_dead_tup from pg_stat_all_tables where relname = 'foo';
 relname | n_dead_tup
---------+------------
 foo     |          3
(1 row)
```

Using VACUUM

```
test=# VACUUM foo;
VACUUM
test=# select relname, n_dead_tup from pg_stat_all_tables where relname = 'foo';
 relname | n_dead_tup
---------+-------------
 foo     |          0
(1 row)
```

Visibility Map

Vacuum maintains a visibility map for each table to keep track of which pages contain only tuples that are known to be visible to all active transactions (and all future transactions, until the page is again modified).

This has two purposes.

vacuum itself can skip such pages on the next run, since there is nothing to clean up.

Second, it allows PostgreSQL to answer some queries using only the index, without reference to the underlying table.

Since PostgreSQL indexes don't contain tuple visibility information, a normal index scan fetches the heap tuple for each matching index entry, to check whether it should be seen by the current transaction. **An index-only scan, on the other hand, checks the visibility map first.** If it's known that all tuples on the page are visible, the heap fetch can be skipped. This is most useful on large data sets where the visibility map can prevent disk accesses.

The visibility map is vastly smaller than the heap, so it can easily be cached even when the heap is very large.

Wrap Around XIDs

PostgreSQL's MVCC transaction semantics depend on being able to compare transaction ID (XID) numbers: a row version with an insertion XID greater than the current transaction XID is “in the future” and should not be visible to the current transaction.

XIDs have limited size of 32 bits so a cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound

XID counter wraps around to zero

transactions that were in the past appear to be in the future — which means their output become invisible. In short, catastrophic data loss.

To avoid this, it is **necessary to vacuum every table in every database at least once every two billion transactions**.

Caveats

Plain **VACUUM** (without FULL) simply reclaims space and makes it available for re-use.

> This form of the command can operate in parallel with normal reading and writing of the table, as an exclusive lock is not obtained.
>
> However, extra space is not returned to the operating system (in most cases); it's just kept available for re-use within the same table.

It also allows us to leverage multiple CPUs in order to process indexes.

> This feature is known as parallel vacuum.

VACUUM FULL rewrites the entire contents of the table into a new disk file with no extra space, allowing unused space to be returned to the operating system.

> This form is much slower and requires an ACCESS EXCLUSIVE lock on each table while it is being processed.

Autovacuum

PostgreSQL has an optional but highly recommended feature called **autovacuum**, whose purpose is to automate the execution of VACUUM and ANALYZE commands.

```
test=# SHOW autovacuum;
 autovacuum
------------
 on
(1 row)
```

Don’t forget

REINDEX

CLUSTER

VACUUM FULL

pg_repack

Transaction ID Wraparound

32-bit transaction ID - Much Too Small

XIDs can be viewed as lying on a circle or circular buffer. As long as the end of that buffer does not jump past the front, the system will function correctly.

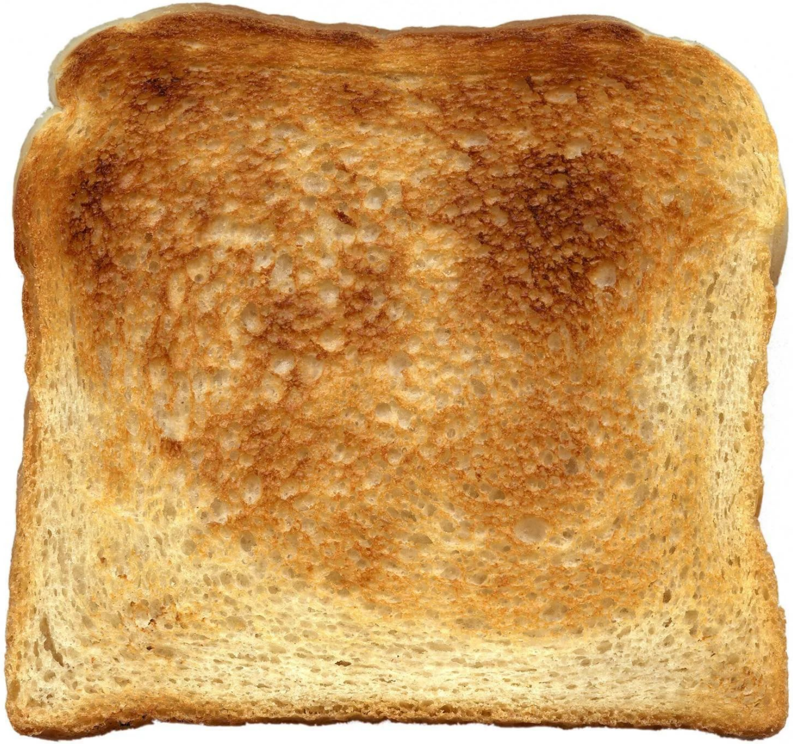
To prevent running out of XIDs and avoid wraparound, the vacuum process is also responsible for “freezing” row versions that are over a certain age (tens of millions of transactions old by default).

However, there are failure modes which prevent it from freezing extremely old tuples and the oldest unfrozen tuple limits the number of past IDs that are visible to a transaction (only two billion past IDs are visible).

If the remaining XID count reaches one million, the database will stop accepting commands and must be restarted in single-user mode to recover. Therefore, it is extremely important to monitor the remaining XIDs so that your database never gets into this state.

TOAST

The Oversized-Attribute Storage Technique – similar to what InnoDB does



Teach Roles

Yes, MySQL has roles but they are not that popular.

PostgreSQL Basics: Roles and Privileges

https://www.red-gate.com/simple-talk/databases/postgresql/postgresql-basics-roles-and-privileges/

PostgreSQL Basics: Object Ownership and Default Privileges

https://www.red-gate.com/simple-talk/uncategorized/postgresql-basics-object-ownership-and-default-privileges/

Wow Factor

The Things a MySQL DBA will be impressed by

Materialized Views, Watch, Many Types of Indexes

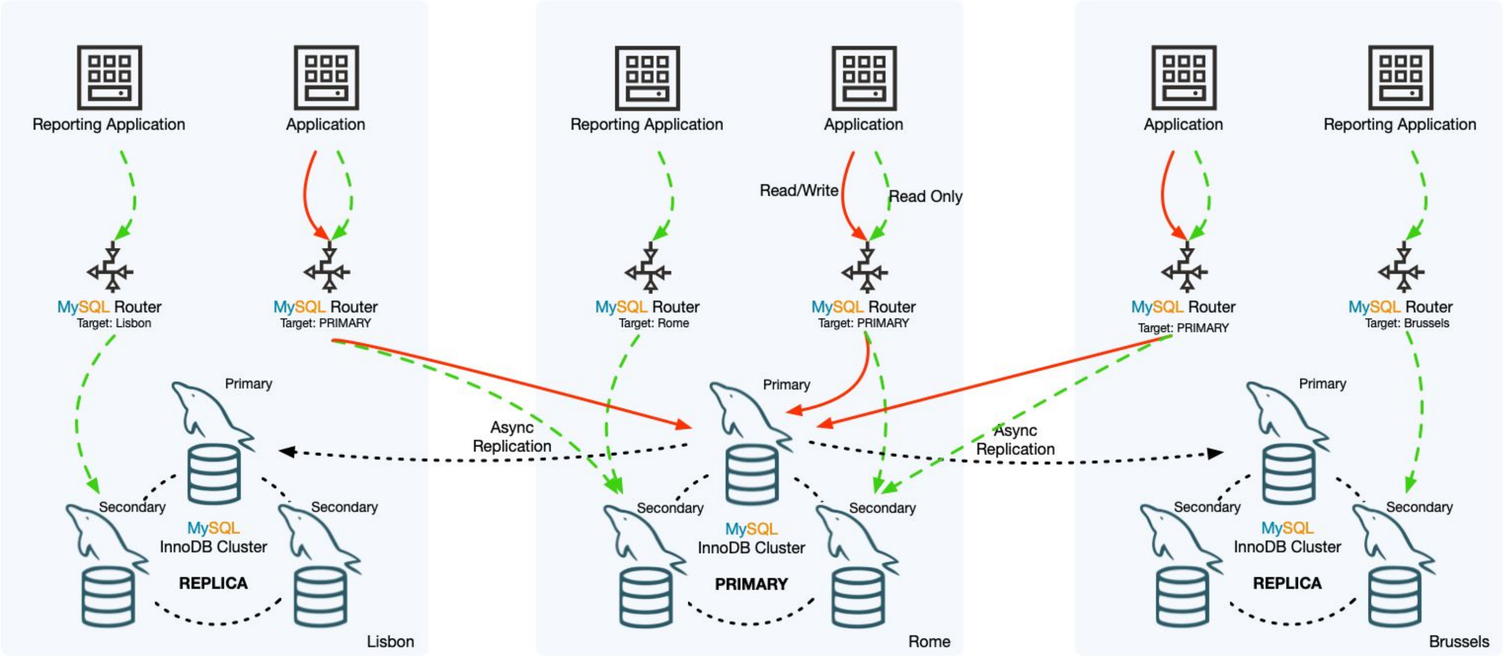
```sql
SELECT
  fa.actor_id,
  SUM(length) FILTER (WHERE rating = 'R'),
  SUM(length) FILTER (WHERE rating = 'PG')
FROM film_actor AS fa
LEFT JOIN film AS f
  ON f.film_id = fa.film_id
GROUP BY fa.actor_id
```

OMGHDWSHTPI2023*

Oh My Goodness How Do We Still Have This Problem In 2023?

Replication

No open source equivalent to InnoDB Cluster or even Galera



Need for connection pooling - multi-process versus multi-threading

Some reading

https://www.youtube.com/watch?v=S7jEJ9o9o2o

https://www.highgo.ca/2021/03/20/how-to-check-and-resolve-bloat-in-postgresql/

https://onesignal.com/blog/lessons-learned-from-5-years-of-scaling-postgresql/

https://www.postgresql.org/docs/

https://www.scalingpostgres.com/

https://psql-tips.org/psql_tips_all.html

“It is different”

Different != Better

What Else To Teach?!?

I *really* need *your* feedback here!

Thank You!

David.Stokes@Percona.com
@Stoker
Speakerdeck.com/Stoker