dbi services by Sequotech

Escaping a public cloud using logical replication with minimal downtime

Who we are

The Company

> Founded in 2010

> More than 100 employees

> Specialized in the Middleware Infrastructure

  > The invisible part of IT

> Customers in Switzerland and all over Europe

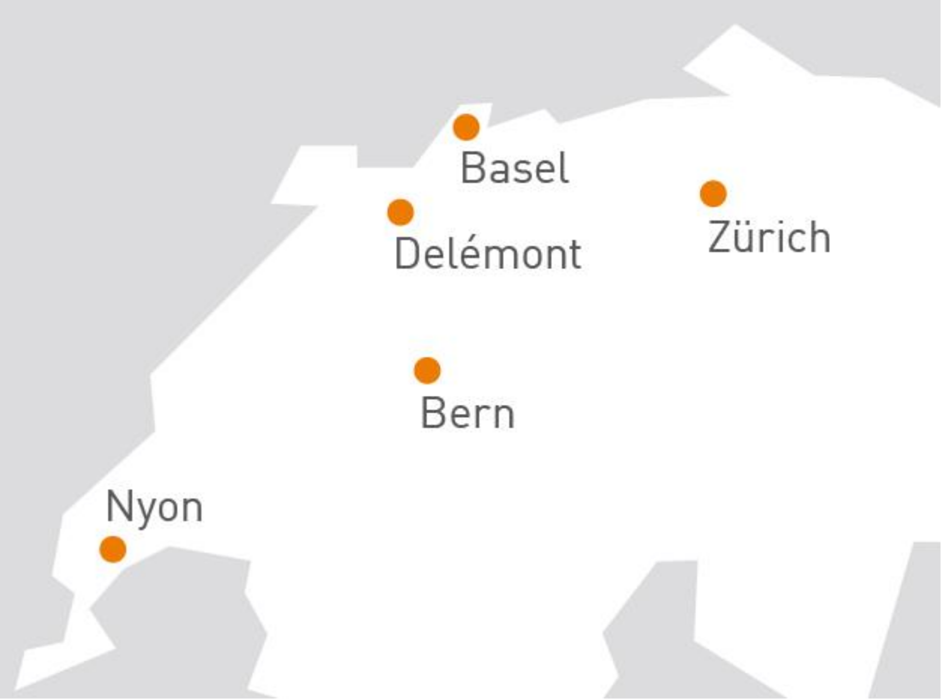
Our Offer

> Consulting

> Service Level Agreements (SLA)

> Trainings

> License Management



swiss safety center ISO 20000 certified system

swiss safety center ISO 27001 certified system

About me

Daniel Westermann

Principal Consultant

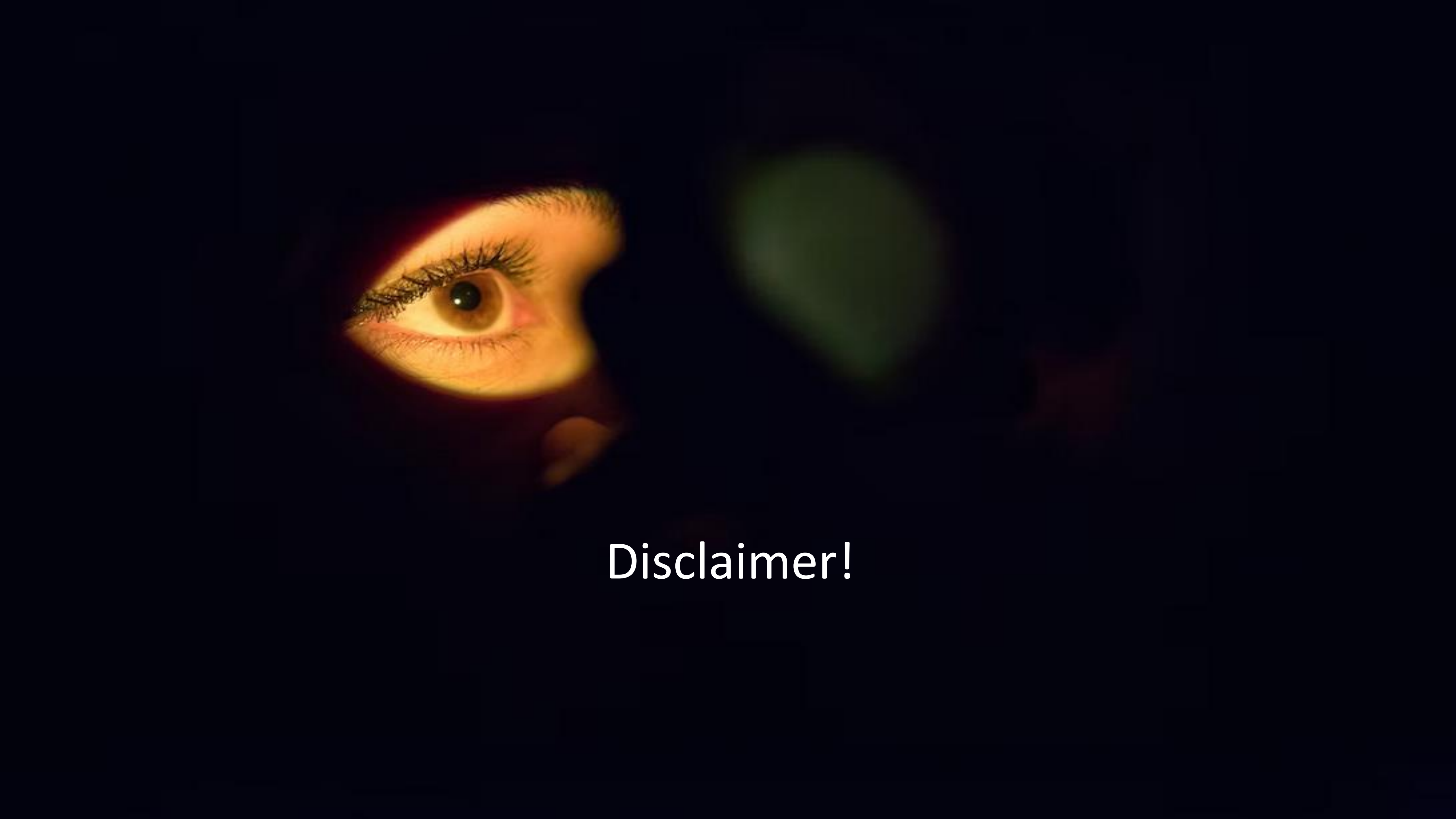
Technology Leader Open Infrastructure

+41 79 927 2446

daniel.westermann[at]dbi-services.com

https://www.linkedin.com/in/daniel-westermann/

@danielwestermann@mastodon.social



Disclaimer!

Disclaimer

What follows is not …

- A recommendation to leave a public cloud
- Blaming of a public cloud provider
- A recommendation to not use a managed service in the cloud

What follows is …

- Know your use case
- Know the public cloud managed services
  - Pricing
  - Flexibility
  - Fallback scenarios
    - How to get out, if required for any reason

This is the story of a customer project

(potential new) customer called

How it started

Initial request

Customer has a customer in a public cloud

- To save money and resources a project started in a public cloud
- Focus was on
  - Getting it up and running as fast as possible
  - Focus on development
  - Easy handling of resources
- No real DBA around
- Mostly a development company
- Used the managed PostgreSQL service of that public cloud provider

How it started
Initial request

A few months after go live

- Storage consumption was at 8TB for production
  - + 8TB for the replica
  - + 2TB for every development clone
- No possibility to archive old data
  - Legal constraints on what can be deleted
  - Even if they could, there is no way to shrink the storage for the managed PostgreSQL service
- Stuck on PostgreSQL 11.x
  - Will go out of support this November

How it started
Initial request

Key pain points to resolve

> Reduce storage consumption

> Define an archival strategy

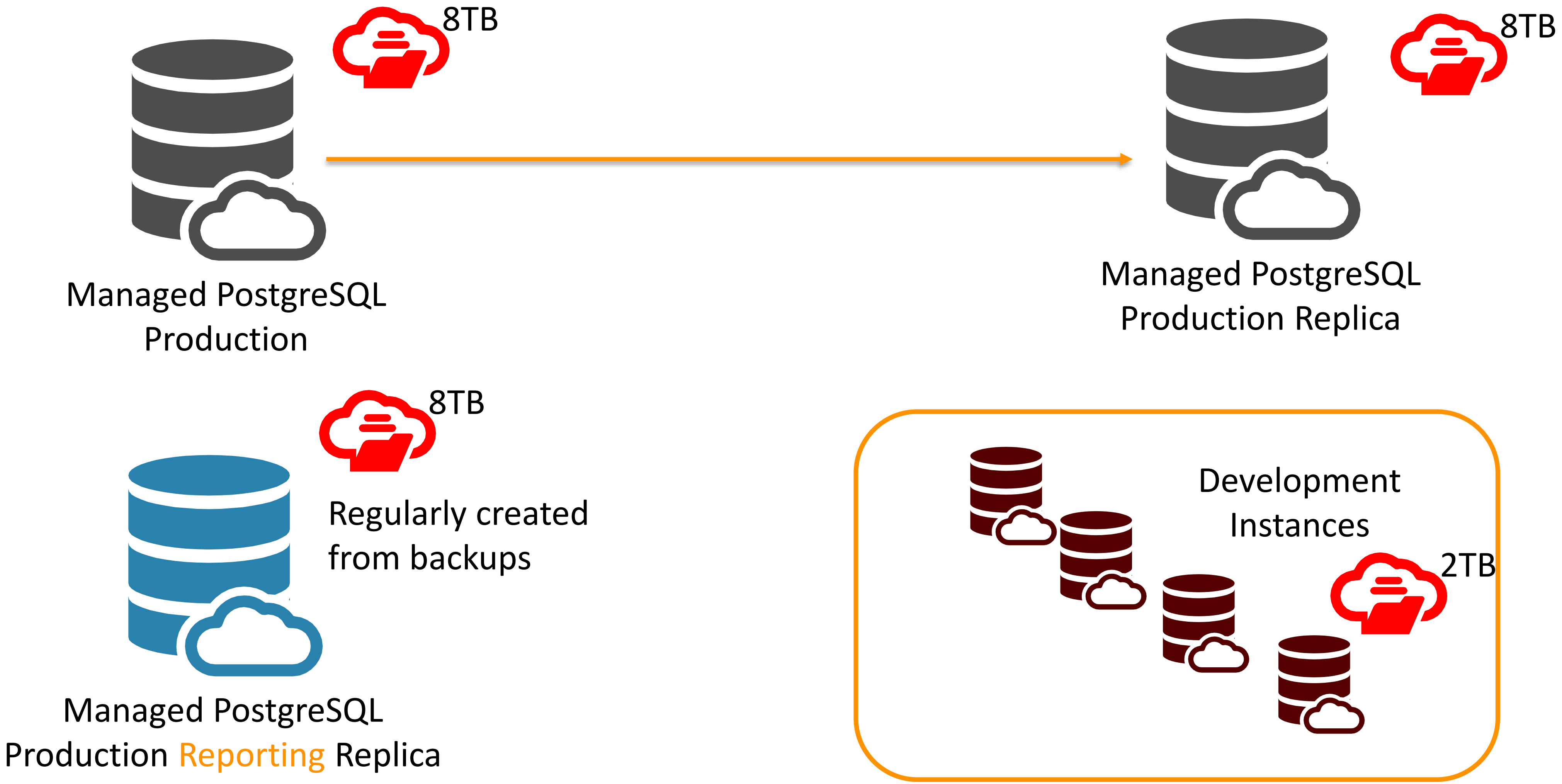
> Upgrade to a more recent version of PostgreSQL

It was all about reducing costs ...

... and give more flexibility

Architecture overview

The initial landscape



8TB

Managed PostgreSQL
Production

8TB

Managed PostgreSQL
Production Replica

8TB

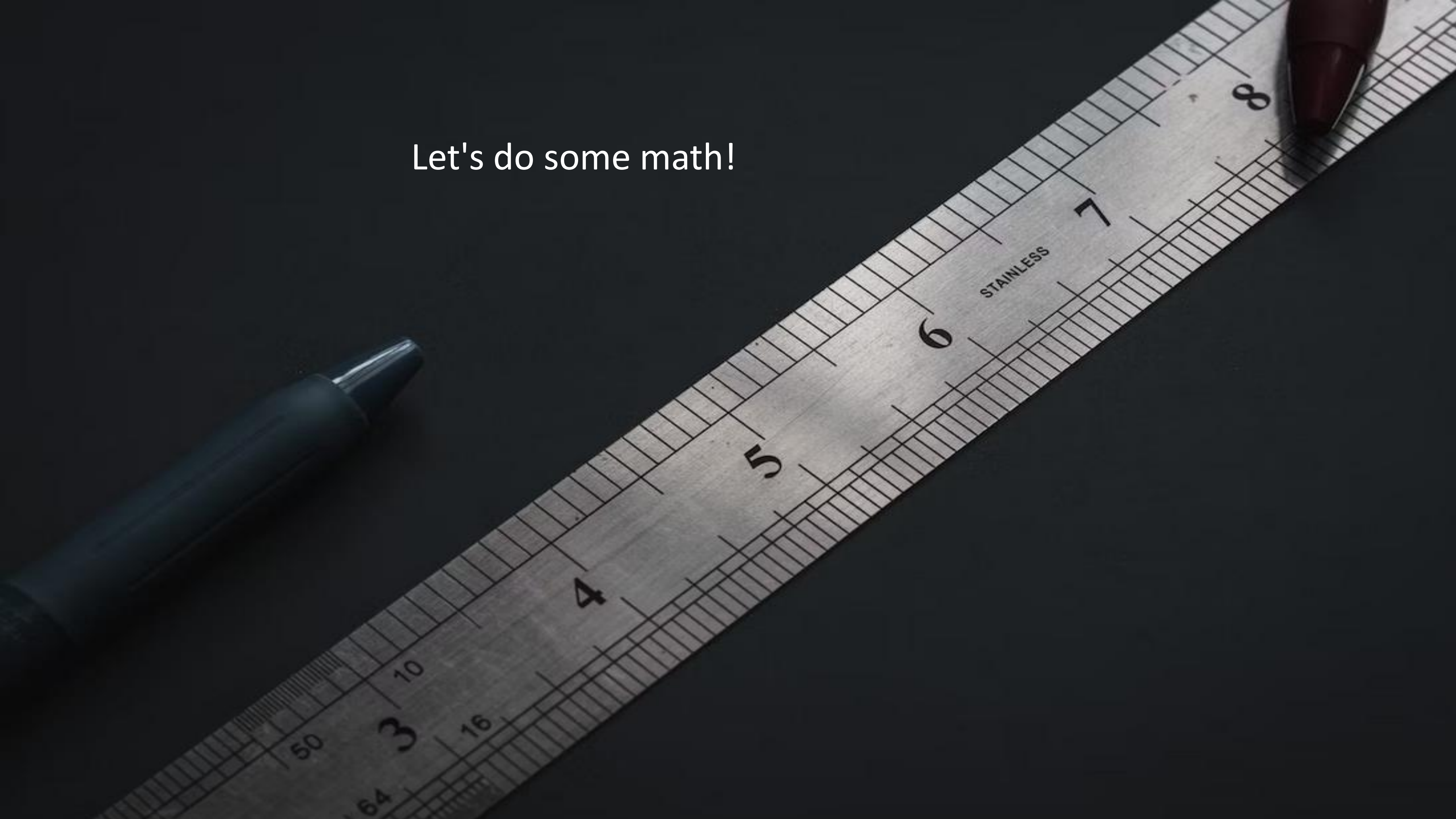
Regularly created
from backups

Managed PostgreSQL
Production Reporting Replica

Development
Instances

2TB

Let's do some math!



Architecture overview
Storage pricing

We'll take 2000 USD per 8TB per month (approx. the average of the three main providers)

- Production: 4000 USD per month
- Reporting: 2000 USD per month
- Development: 1000 USD per month
- Backup storage: 2500 USD per month (half the price)
- 9500 USD overall -> 114'000 per year, just for the storage
  - This does not include compute and network costs

- This is per end-customer of the customer's customer
  - Yes, things can get complicated

Priority 1: Reduce storage consumption

Options

Priority one: Storage reduction

What options do we have to reduce storage consumption?

- vacuum full?
  - This is a blocking operation
- Getting rid of old data?
  - Create an archival strategy
- Optimize how PostgreSQL stores data?
- Compression?
- Getting rid of unused / redundant indexes?

What do all these options do have in common?

- They will not reduce the costs associated to the storage in a public cloud
  - None of the major public cloud providers offers a way to reduce the size of volumes

Options

Priority one: Storage reduction

What options do we have to reduce storage consumption?

- vacuum full?
  - This is a blocking operation
- Getting rid of old data?
  - Create an archival strategy
- Optimize how PostgreSQL stores data?
- Compression?
- Getting rid of unused / redundant indexes?

What do all these options do have in common?

- They will not reduce the costs associated to the storage in a public cloud
  - None of the major public cloud providers offers a way to reduce the size of volumes

Options

Priority one: Storage reduction

Alignment padding

> An empty row in PostgreSQL

```
postgres=# SELECT pg_column_size(row()) as bytes;
 bytes
-------
    24
(1 row)
```

> One SMALLINT column

```
postgres=# SELECT pg_column_size(row(0::smallint)) as bytes;
 bytes
-------
    26
(1 row)
```

Options

Priority one: Storage reduction

Alignment padding

- One BIGINT column

```
postgres=# SELECT pg_column_size(row(0::bigint)) as bytes;
 bytes
-------
    32
(1 row)
```

- So what?

```
postgres=# SELECT pg_column_size(row(0::smallint,0::bigint)) as bytes;
 bytes
-------
    40
(1 row)
```

- ?? 2 + 8 = 16?

Options

Priority one: Storage reduction

Alignment padding

> The internal alignment in PostgreSQL is 8 bytes

> Fixed length columns that follow each other must be padded with empty bytes in some cases

> Instead of 2+8 the math becomes 8+8

```
postgres=# SELECT pg_column_size(row(0::smallint,0::bigint)) as bytes;
 bytes
-------
    40
(1 row)
```

Options

Priority one: Storage reduction

Alignment padding

> Given this simple table

```
postgres=# create table t ( a boolean, b smallint, c timestamp, d smallint, e bigint );
CREATE TABLE
postgres=# \d t
                           Table "public.t"
 Column |            Type             | Collation | Nullable | Default
--------+-----------------------------+-----------+----------+---------
 a      | boolean                     |           |          |
 b      | smallint                    |           |          |
 c      | timestamp without time zone |           |          |
 d      | smallint                    |           |          |
 e      | bigint                      |           |          |
```

Options

Priority one: Storage reduction

Alignment padding

> This is what PostgreSQL knows / sees

```
postgres=# SELECT a.attname, t.typname, t.typalign, t.typlen
             FROM pg_class c
             JOIN pg_attribute a ON (a.attrelid = c.oid)
             JOIN pg_type t ON (t.oid = a.atttypid)
            WHERE c.relname = 't'
              AND a.attnum >= 0
            ORDER BY a.attnum;
 attname |  typname  | typalign | typlen
---------+-----------+----------+--------
 a       | bool      | c        |      1
 b       | int2      | s        |      2
 c       | timestamp | d        |      8
 d       | int2      | s        |      2
 e       | int8      | d        |      8
(5 rows)
```

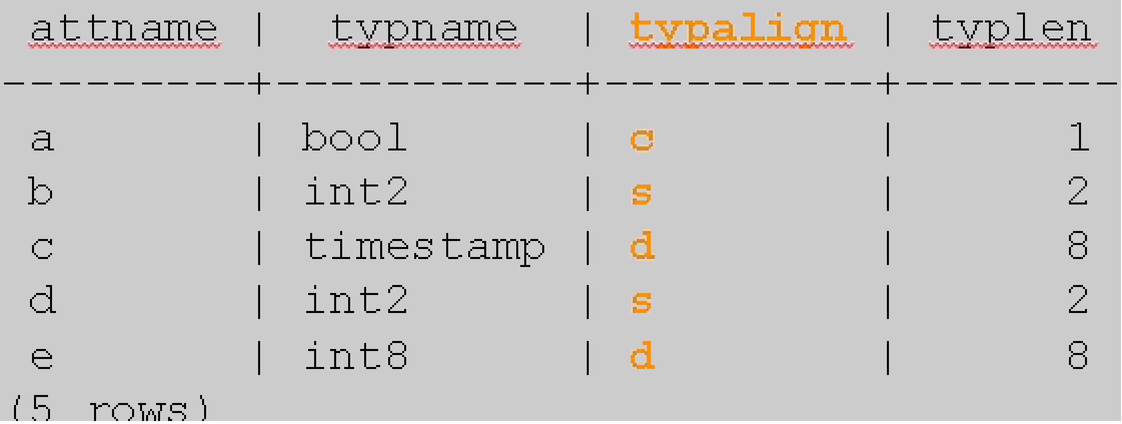
Options

Priority one: Storage reduction

Alignment padding

> https://www.postgresql.org/docs/current/catalog-pg-type.html

Value	Meaning
c	char alignment, no alignment needed
s	short alignment (2 bytes)
i	int alignment (4 bytes)
d	double alignment (8 bytes)



```
 attname |  typname  | typalign | typlen
---------+-----------+----------+--------
 a       | bool      | c        |      1
 b       | int2      | s        |      2
 c       | timestamp | d        |      8
 d       | int2      | s        |      2
 e       | int8      | d        |      8
(5 rows)
```

Options

Priority one: Storage reduction

Creating one million rows in that table

```
postgres=# insert into t
           select true
                , 1
                , now()
                , 1
                , i
             from generate_series(1,1000000) i;
INSERT 0 1000000
postgres=# select pg_size_pretty(pg_relation_size('t'));
 pg_size_pretty
----------------
 57 MB
(1 row)
```

Options

Priority one: Storage reduction

Fixing the column order

```
postgres=# drop table t;
DROP TABLE
postgres=# create table t ( e bigint, c timestamp, b smallint, d smallint, a boolean );
CREATE TABLE
postgres=# insert into t select i
                , now()
                , 1
                , 1
                , true from generate_series(1,1000000) i;
INSERT 0 1000000
postgres=# select pg_size_pretty(pg_relation_size('t'));
 pg_size_pretty
----------------
 50 MB
(1 row)
```

> This saved 7MB of overhead!

Options

Priority one: Storage reduction

The rule for column ordering is

- Large fix sized columns at the beginning
  - e.g. BIGINT, TIMESTAMP
- Smaller fixed sized columns after, in decending order of the the size
  - e.g. INT then SMALLINT ...
- Variable length columns at the end
  - e.g. NUMERIC, TEXT

Options

Priority one: Storage reduction

Getting the correct column ordering out of the catalog

```
postgres=# SELECT a.attname, t.typname, t.typalign, t.typlen
             FROM pg_class c
             JOIN pg_attribute a ON (a.attrelid = c.oid)
             JOIN pg_type t ON (t.oid = a.atttypid)
            WHERE c.relname = 't'
              AND a.attnum >= 0
            ORDER BY t.typlen DESC;
 attname |  typname  | typalign | typlen
---------+-----------+----------+--------
 e       | int8      | d        |      8
 c       | timestamp | d        |      8
 b       | int2      | s        |      2
 d       | int2      | s        |      2
 a       | bool      | c        |      1
(5 rows)
```

Options

Priority one: Storage reduction

We've tested that on one of the test environments

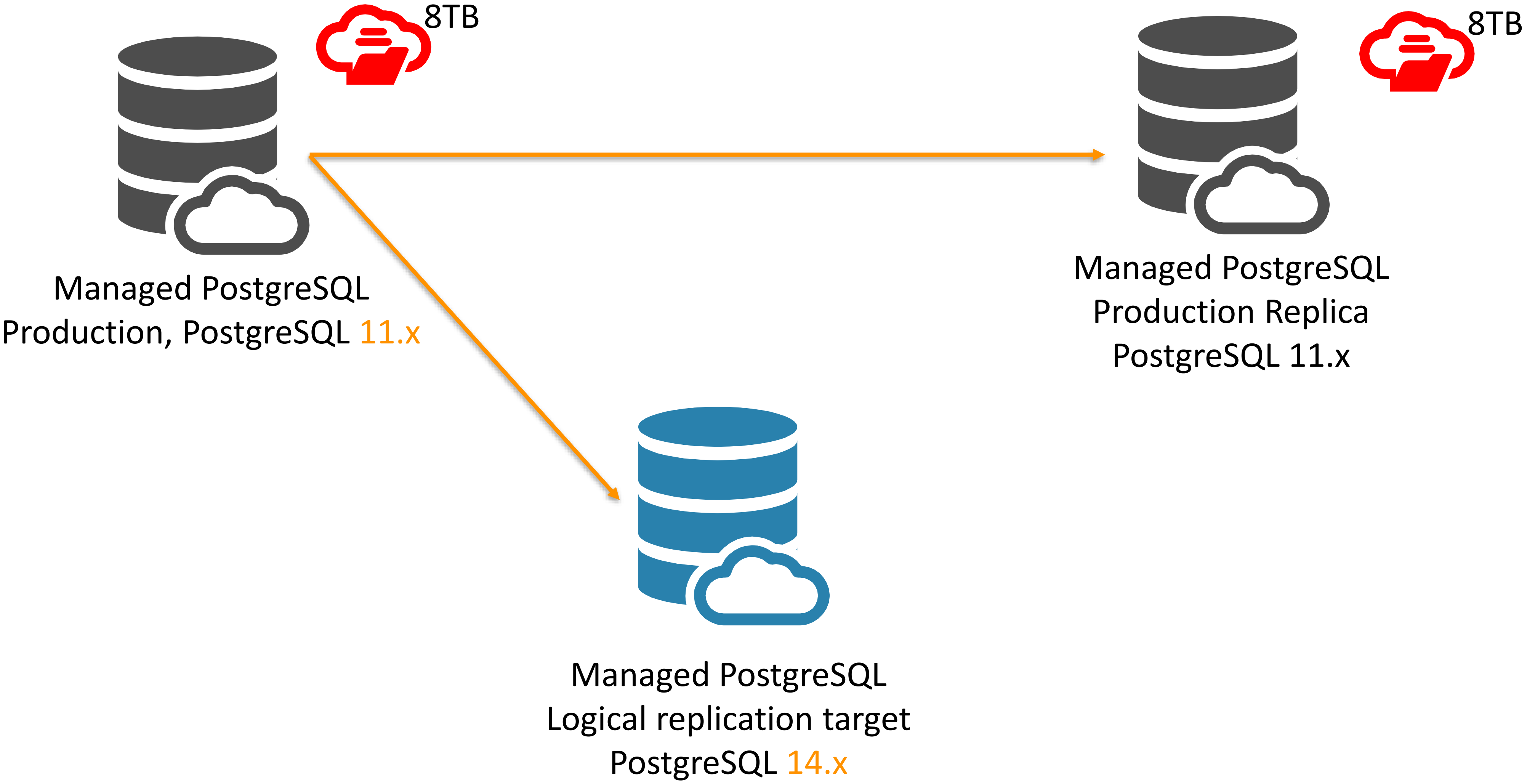
- Only fixing the column order resulted in 11% storage reduction
  - This is 880 GB per instance!
  - Of course changing the column order could force application level changes as well

The only options to implement this?

- Create a new instance
  - pg_dump / pg_restore
    - Problem: Downtime
- Create a new instance
  - Create the schema with the correct ordering of the columns
  - Setup logical replication
    - Problem: The initial load will take some time
    - Problem: Sequences are not replicated
- Both solutions will temporarily increase the storage costs

The new setup

Setting up logical replication



A bit of history

PostgreSQL logical replication
A bit of history

PostgreSQL 9.0 (20-SEP-2010) - out of support!

> Physical replication

> Only between the same major versions of PostgreSQL

PostgreSQL 9.6 (29-SEP-2016) - out of support!

> Logical decoding

  > allows extensions to insert data into the WAL stream that can be read by logical-decoding plugins

PostgreSQL logical replication
A bit of history

PostgreSQL 10 (05-OCT-2017) - out of support

- Logical replication
  - Using publish / subscribe
- Restrictions
  - No replication of DDL commands
  - No replication of sequences
  - No replication of TRUNCATE commands
  - No replication of LARGE objects
  - Only from base tables to base tables
    - No views, materialized views, partition root tables, foreign tables
    - In case of partitions only to the same partition structure

PostgreSQL logical replication

A bit of history

PostgreSQL 10 (05-OCT-2017) - out of support

```
postgres=# \h create publication
Command:     CREATE PUBLICATION
Description: define a new publication
Syntax:
CREATE PUBLICATION name
    [ FOR TABLE [ ONLY ] table_name [ * ] [, ...]
      | FOR ALL TABLES ]
    [ WITH ( publication_parameter [= value] [, ... ] ) ]

postgres=#
```

- > For all tables
- > For a list of tables
- > Publication parameters: publish='insert, update, delete'

PostgreSQL logical replication
A bit of history

PostgreSQL 11 (18-OCT-2018) - out of support November 2023

- Logical replication
  - Using publish / subscribe
  - Allow replication slots to be advanced programatically - pg_replication_slot_advance()
- Restrictions
  - No replication of DDL commands
  - No replication of sequences
  - *No replication of TRUNCATE commands - restriction removed*
  - No replication of LARGE objects
  - Only from base tables to base tables
    - No views, materialized views, partition root tables, foreign tables
    - In case of partitions only to the same partition structure

PostgreSQL logical replication

A bit of history

PostgreSQL 11 (18-OCT-2018) - out of support

```
postgres=# \h create publication
Command:     CREATE PUBLICATION
Description: define a new publication
Syntax:
CREATE PUBLICATION name
    [ FOR TABLE [ ONLY ] table_name [ * ] [, ...]
      | FOR ALL TABLES ]
    [ WITH ( publication_parameter [= value] [, ... ] ) ]

postgres=#
```

> For all tables

> For a list of tables

> Publication parameters: publish='insert, update, delete, **truncate**'

PostgreSQL logical replication

A bit of history

PostgreSQL 12 (03-OCT-2019) - out of support November 2024

- Logical replication
  - Using publish / subscribe
  - Allow replication slots to be advanced programatically - pg_replication_slot_advance()
  - Allow relocation slots to be copied - pg_copy_logical_replication_slot()
- Restrictions
  - No replication of DDL commands
  - No replication of sequences
  - No replication of LARGE objects
  - Only from base tables to base tables
    - No views, materialized views, partition root tables, foreign tables
    - In case of partitions only to the same partition structure

PostgreSQL logical replication
A bit of history

PostgreSQL 12 (03-OCT-2019) - out of support November 2024

```
postgres=# \h create publication
Command:     CREATE PUBLICATION
Description: define a new publication
Syntax:
CREATE PUBLICATION name
    [ FOR TABLE [ ONLY ] table_name [ * ] [, ...]
      | FOR ALL TABLES ]
    [ WITH ( publication_parameter [= value] [, ... ] ) ]

postgres=#
```

> For all tables

> For a list of tables

> Publication parameters: publish='insert, update, delete, truncate'

PostgreSQL logical replication

A bit of history

PostgreSQL 13 (24-SEP-2020)

- Logical replication
  - Allow replication slots to be advanced programatically - pg_replication_slot_advance()
  - Allow relocation slots to be copied - pg_copy_logical_replication_slot()
  - Allow partitioned tables to be replicated, not only the individual partitions
  - Allow logical replication into partitioned tables on the subscriber
  - Allow control over how much memory is used by logical decoding - logical_decoding_work_mem
- Restrictions
  - No replication of DDL commands
  - No replication of sequences
  - No replication of LARGE objects
  - Only from base tables to base tables
    - No views, materialized views, ~~partition root tables,~~ foreign tables
    - ~~In case of partitions only to the same partition structure~~

PostgreSQL logical replication

A bit of history

PostgreSQL 13 (24-SEP-2020)

```
postgres=# \h create publication
Command:     CREATE PUBLICATION
Description: define a new publication
Syntax:
CREATE PUBLICATION name
    [ FOR TABLE [ ONLY ] table_name [ * ] [, ...]
      | FOR ALL TABLES ]
    [ WITH ( publication_parameter [= value] [, ... ] ) ]
```

- For all tables
- For a list of tables
- Publication parameters:
  - publish='insert, update, delete, truncate'
  - publish_via_partition_root=true/false

PostgreSQL logical replication

A bit of history

PostgreSQL 14 (30-SEP-2021)

- Logical replication
  - Allow replication slots to be advanced programatically - pg_replication_slot_advance()
  - Allow relocation slots to be copied - pg_copy_logical_replication_slot()
  - Allow partitioned tables to be replicated, not only the individual partitions
  - Allow logical replication into partitioned tables on the subscriber
  - Allow control over how much memory is used by logical decoding - logical_decoding_work_mem
  - Allow streaming of long in-progress transactions
  - Various performance improvements
- Restrictions
  - No replication of DDL commands
  - No replication of sequences
  - No replication of LARGE objects
  - Only from base tables to base tables
    - No views, materialized views, foreign tables

PostgreSQL logical replication
A bit of history

PostgreSQL 14 (24-SEP-2021)

```
postgres=# \h create publication
Command:     CREATE PUBLICATION
Description: define a new publication
Syntax:
CREATE PUBLICATION name
    [ FOR TABLE [ ONLY ] table_name [ * ] [, ...]
      | FOR ALL TABLES ]
    [ WITH ( publication_parameter [= value] [, ... ] ) ]
```

- For all tables
- For a list of tables
- Publication parameters:
  - publish='insert, update, delete, truncate'
  - publish_via_partition_root=true/false

PostgreSQL logical replication

A bit of history

PostgreSQL 15 (13-OCT-2022)

- Logical replication
  - Allow replication slots to be advanced programatically - pg_replication_slot_advance()
  - Allow relocation slots to be copied - pg_copy_logical_replication_slot()
  - Allow partitioned tables to be replicated, not only the individual partitions
  - Allow logical replication into partitioned tables on the subscriber
  - Allow control over how much memory is used by logical decoding - logical_decoding_work_mem
  - Allow streaming of long in-progress transactions
  - Various performance improvements
  - Allow selective publication
    - Column lists and filter conditions
- Restrictions
  - No replication of DDL commands
  - No replication of sequences
  - No replication of LARGE objects

PostgreSQL logical replication

A bit of history

PostgreSQL 15 (13-OCT-2022)

```
postgres=# \h create publication
Command:     CREATE PUBLICATION
Description: define a new publication
Syntax:
CREATE PUBLICATION name
    [ FOR ALL TABLES
      | FOR publication_object [, ... ] ]
    [ WITH ( publication_parameter [= value] [, ... ] ) ]

where publication_object is one of:

    TABLE [ ONLY ] table_name [ * ] [ ( column_name [, ... ] ) ] [ WHERE ( expression ) ]
[, ... ]
    TABLES IN SCHEMA { schema_name | CURRENT_SCHEMA } [, ... ]
```

> For all tables

> Column lists and where conditions

PostgreSQL logical replication

A bit of history

PostgreSQL 16 (??-??-2023) - currently in Beta - Please test

- Logical replication
  - Allow selective publication
    - Column lists and filter conditions
  - Allow logical replication from replicas
  - Allow logical replication subscribers to apply large transactions in parallel
  - Allow parallel application of logical replication
- Restrictions
  - No replication of DDL commands
  - No replication of sequences
  - No replication of LARGE objects

PostgreSQL logical replication

A bit of history

PostgreSQL 16 (??-??-2023) - currently in Beta - Please test

```
postgres=# \h create publication
Command:     CREATE PUBLICATION
Description: define a new publication
Syntax:
CREATE PUBLICATION name
    [ FOR ALL TABLES
      | FOR publication_object [, ... ] ]
    [ WITH ( publication_parameter [= value] [, ... ] ) ]

where publication_object is one of:

    TABLE [ ONLY ] table_name [ * ] [ ( column_name [, ... ] ) ] [ WHERE ( expression ) ]
[, ... ]
    TABLES IN SCHEMA { schema_name | CURRENT_SCHEMA } [, ... ]
```

- For all tables
- Column lists and where conditions

PostgreSQL logical replication

A bit of history

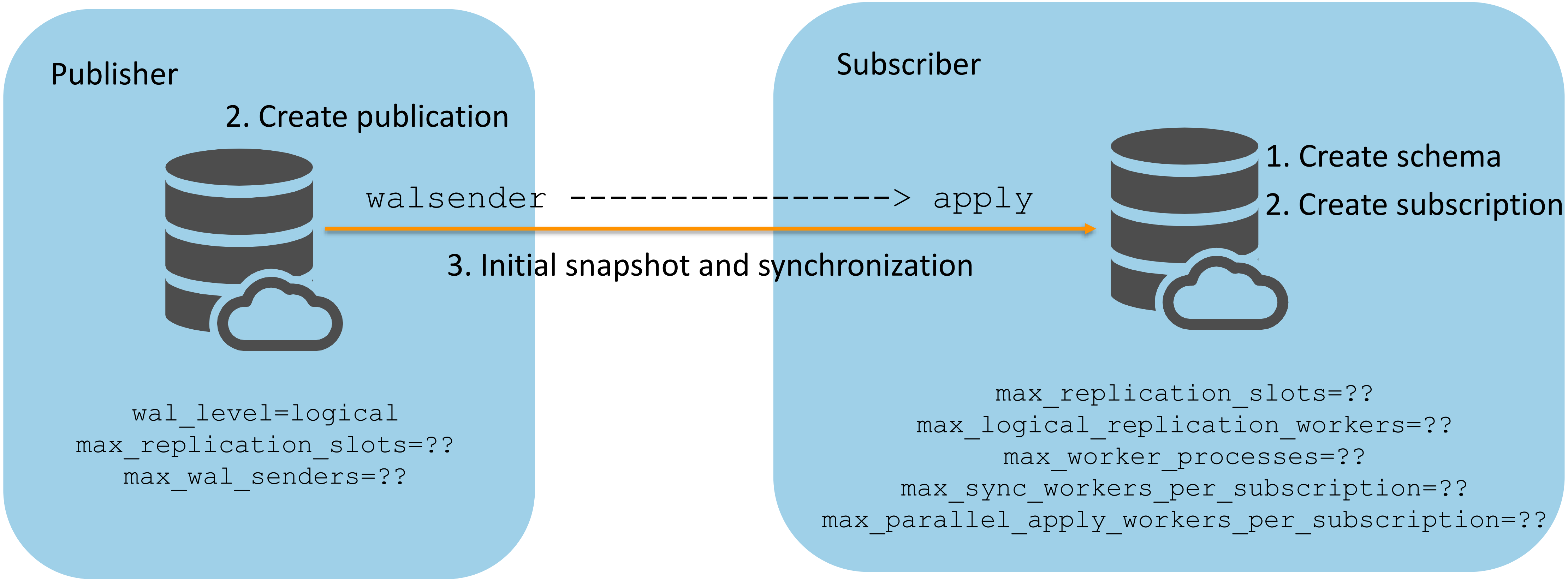
PostgreSQL 17 (??-??-2024) - in development

- Logical replication
  - Allow selective publication
    - Column lists and filter conditions
  - Allow logical replication from replicas
  - Allow logical replication subscribers to apply large transactions in parallel
  - Allow parallel application of logical replication
  - Allow replication of DDLs?
    - https://commitfest.postgresql.org/43/3595/
  - Skip replicating the tables specified in except table option?
    - https://commitfest.postgresql.org/43/3646/
- Restrictions
  - ???

Architecture

Logical replication

Architecture



Publisher

2. Create publication

```
walsender -----------------> apply
```

3. Initial snapshot and synchronization

```
wal_level=logical
max_replication_slots=??
max_wal_senders=??
```

Subscriber

1. Create schema
2. Create subscription

```
max_replication_slots=??
max_logical_replication_workers=??
max_worker_processes=??
max_sync_workers_per_subscription=??
max_parallel_apply_workers_per_subscription=??
```

Logical replication
Architecture

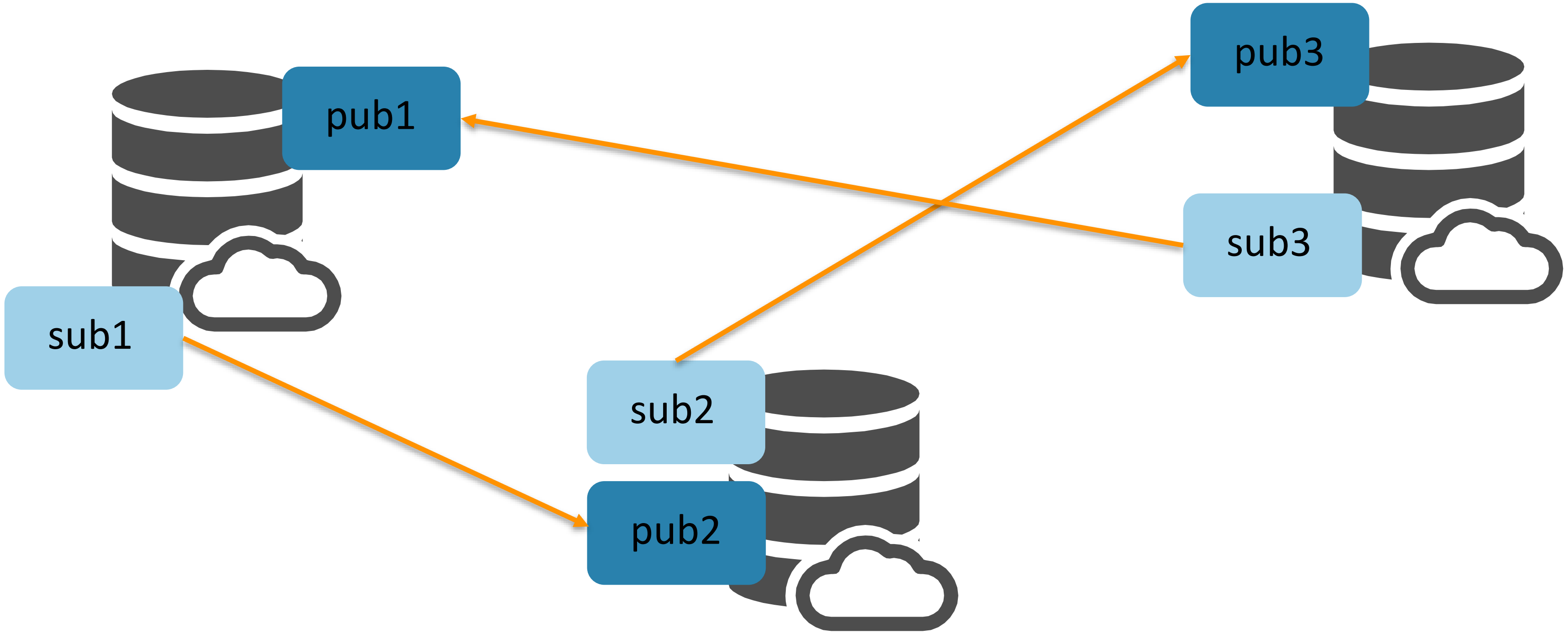
REPLICA IDENTITY

- A table must have a replica identity
  - so rows to be updated and deleted can be identified from the subscriber side
- By default this is the primary key
- Otherwise a unique key should be set
- FULL
  - Indexes can be used to identify the rows or
  - All columns of the table, slower
  - Should not be used

```
postgres=# \h alter table
…
REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }
…
```

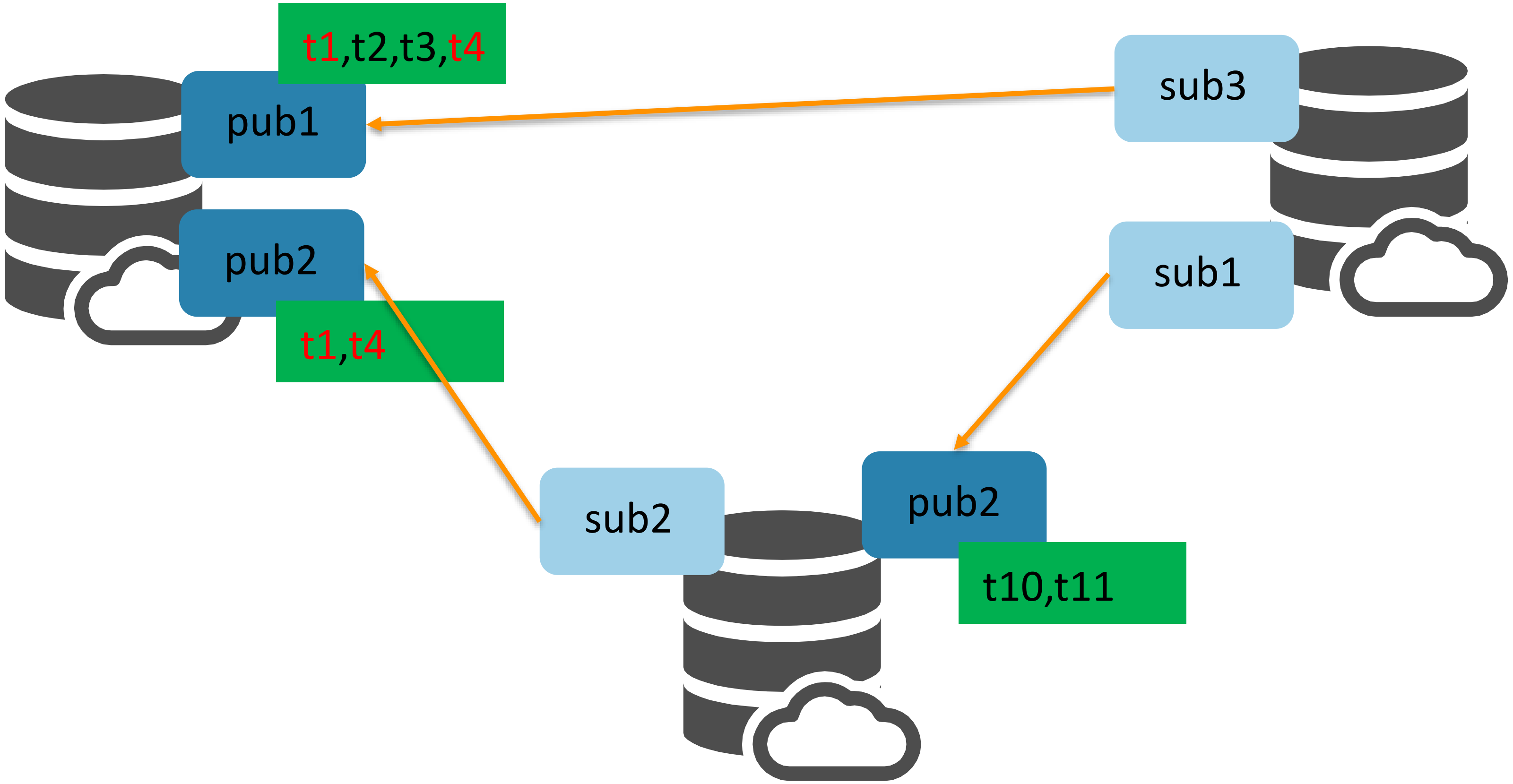
Logical replication
Architecture

A publisher can also be a subscripber, and vice versa



Logical replication
Architecture

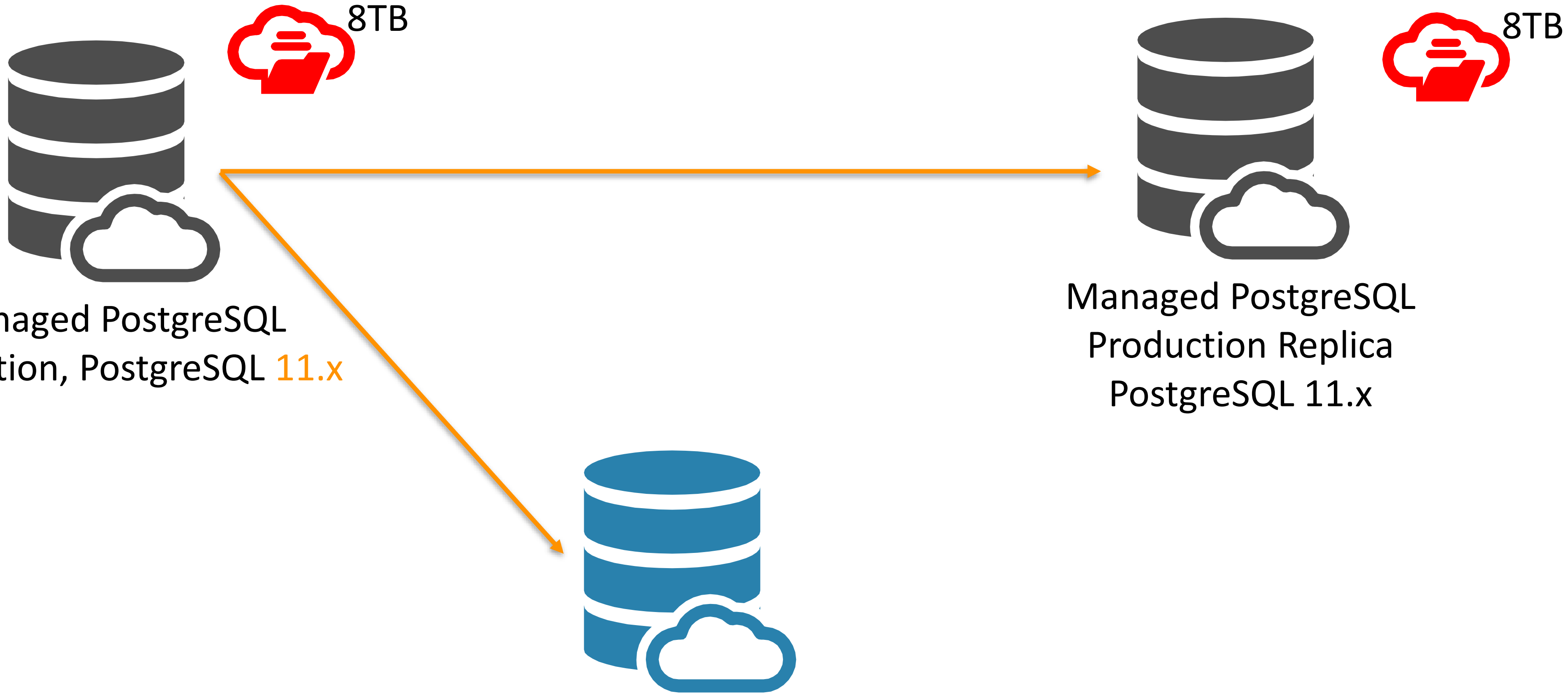
The same table can be in multiple publications



Returning to the setup

The new setup

Setting up logical replication



Managed PostgreSQL
Production, PostgreSQL 11.x

Managed PostgreSQL
Production Replica
PostgreSQL 11.x

Managed PostgreSQL
Logical replication target
PostgreSQL 14.x

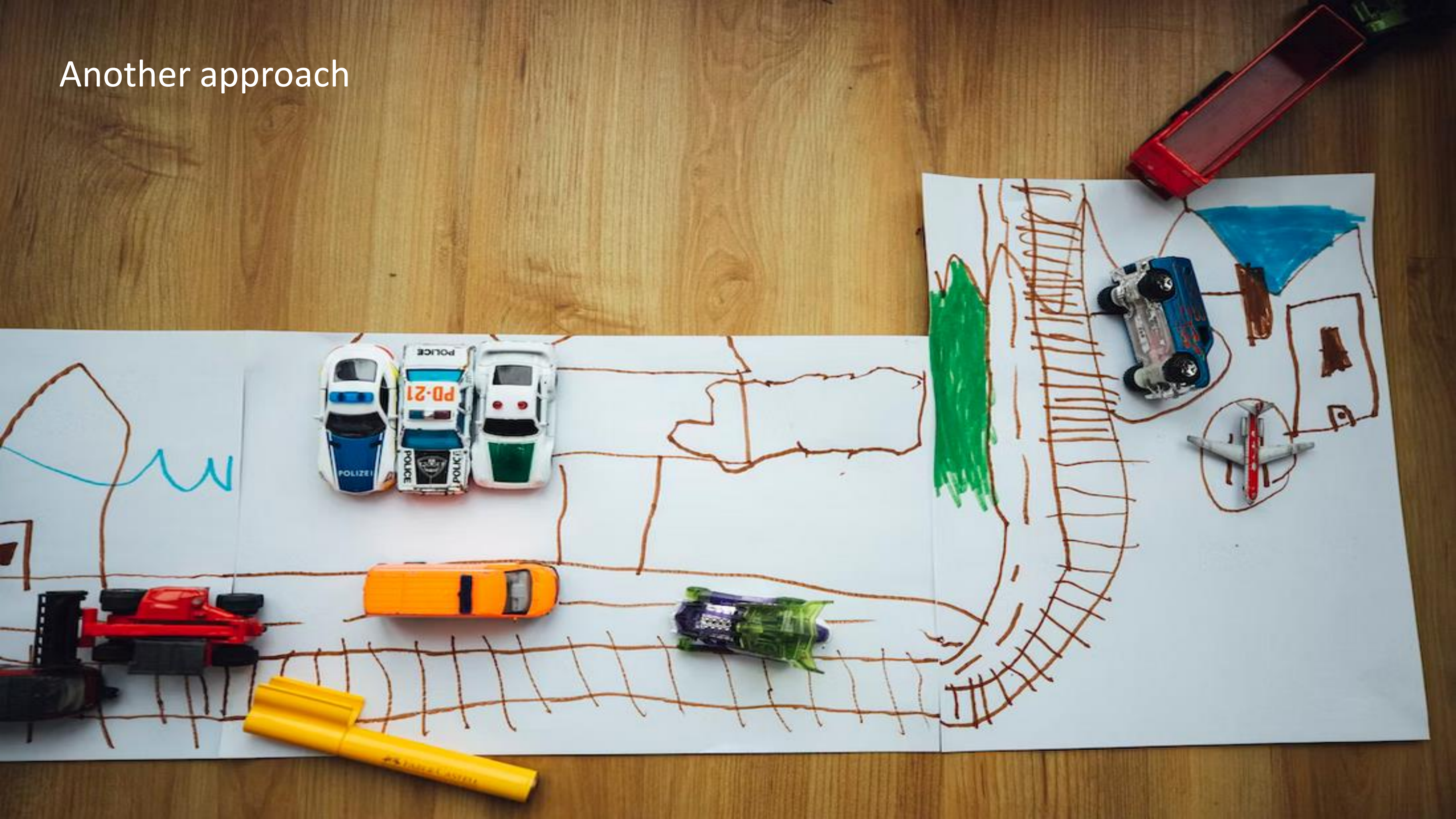
This did not work!

The new setup
Issues

Why this didn't work

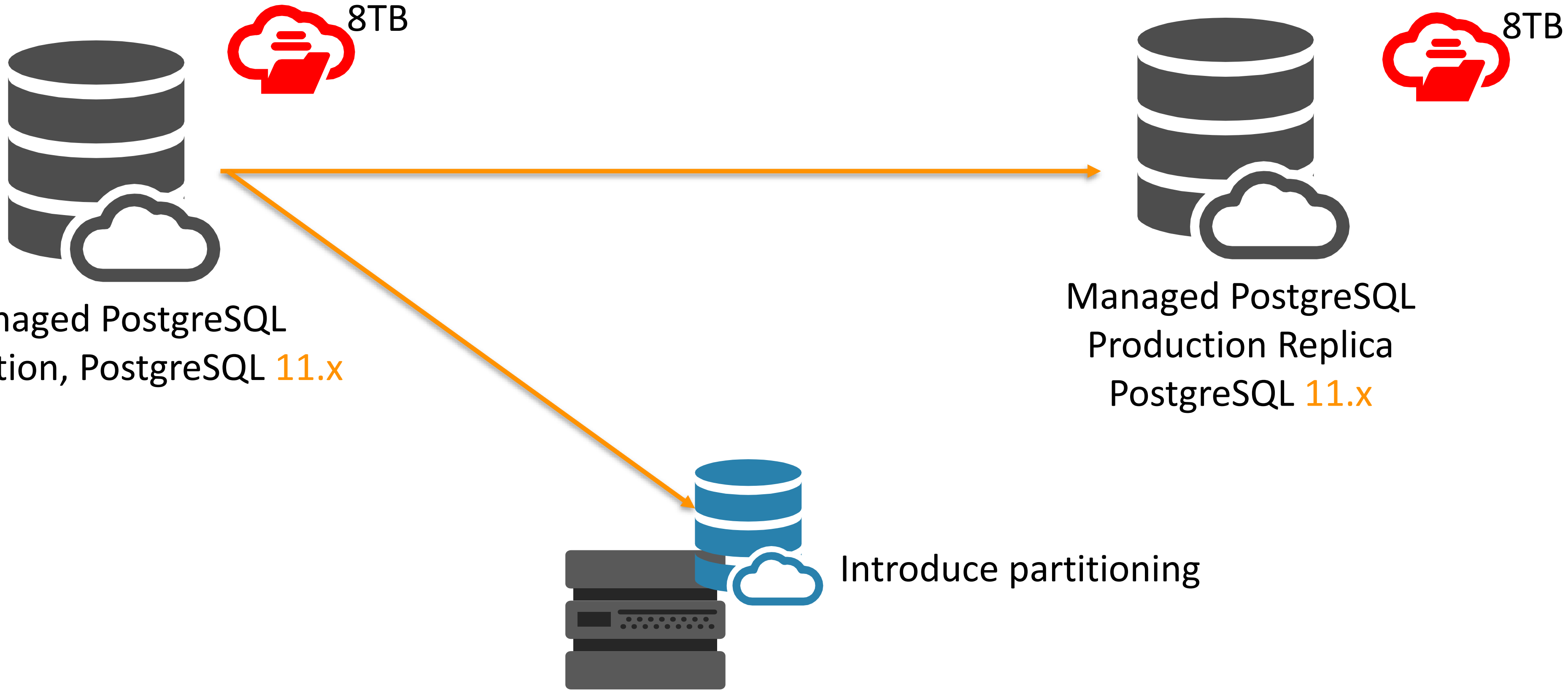
- The initial load was taking more than a week
- For the target, to save costs, cheaper disks have been chosen
  - This slowed down the replication
- The publisher could not remove WAL for a very long time
  - Storage increase
  - Even more costs
- Indexes and primary keys have not been removed on the subscriber
  - More slow down
- More costs for an adittional managed PostgreSQL instance
- Limited insight on what was going on on the operating system
  - You don't have access to that in a managed PostgreSQL cloud service

Another approach



The new setup - take two

Setting up logical replication



Managed PostgreSQL
Production, PostgreSQL 11.x

Managed PostgreSQL
Production Replica
PostgreSQL 11.x

Introduce partitioning

Self-Managed PostgreSQL
Logical replication target
PostgreSQL 15.x

The new setup - take two
Advantages / disadvantages

Why self managed on a VM

- Full control of the operating system
  - I/O statistics
  - Memory
  - Network
- We could use the latest version of PostgreSQL (15)
  - The managed service only offered 14.x
- Faster to scale up and down
  - A VM with PostgreSQL is starting much faster than a managed service
  - Much more flexibility with the storage options
- Comes with the possibility for partitionig
  - Pre-partition the large tables
    - Archive data goes to cheap storage
    - Live data is on fast, but more expensive storage
- Cheaper than the managed service

Did it work?

No!

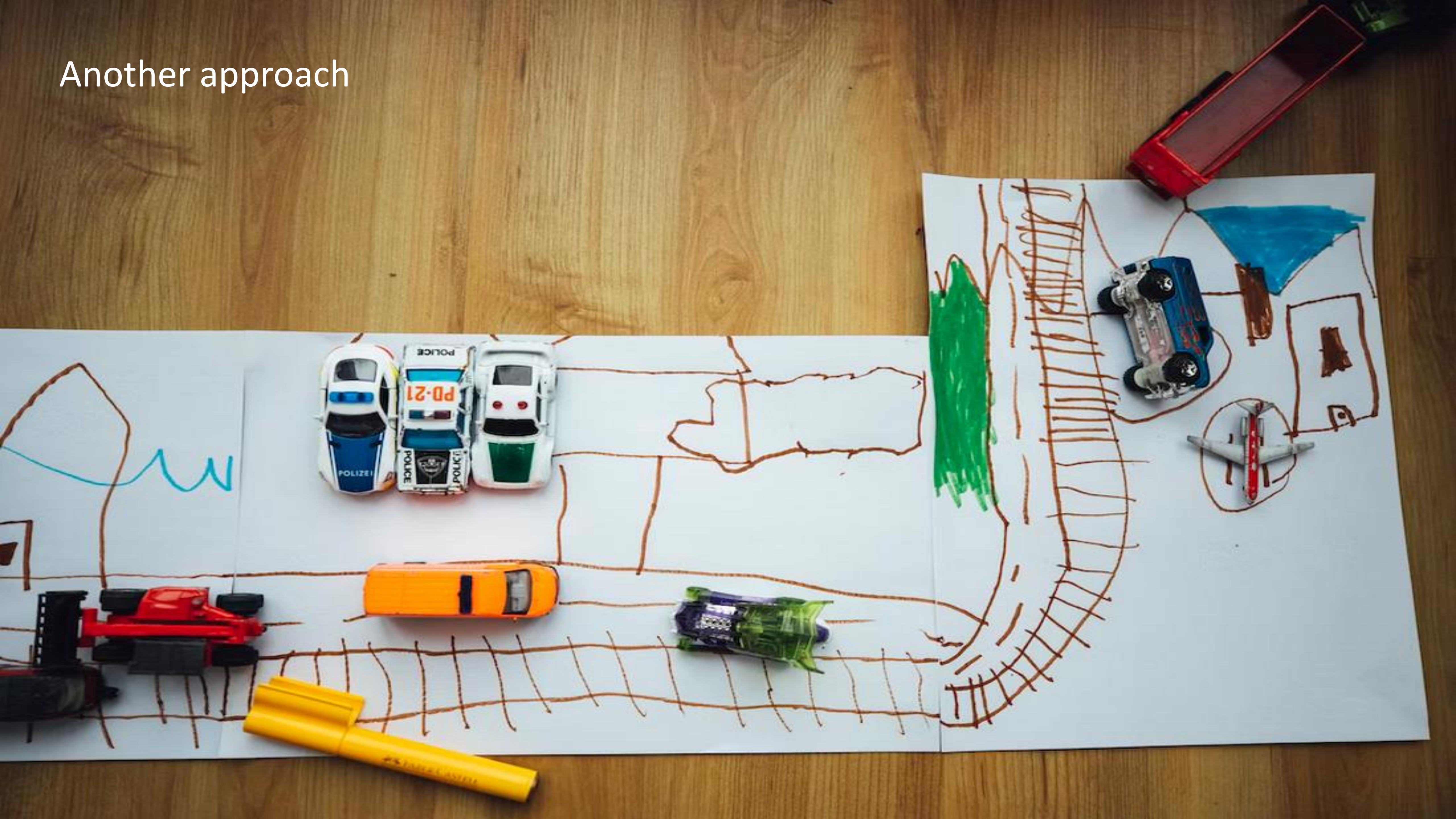
The new setup - take two

Advantages / disadvantages

Why it didn't work as well

- The initial load once more took too long
  - Was stopped after one and a half weeks
- We still had the issue with increasing WAL usage on the publisher
  - More costs for the expensive managed service on the source

Another approach

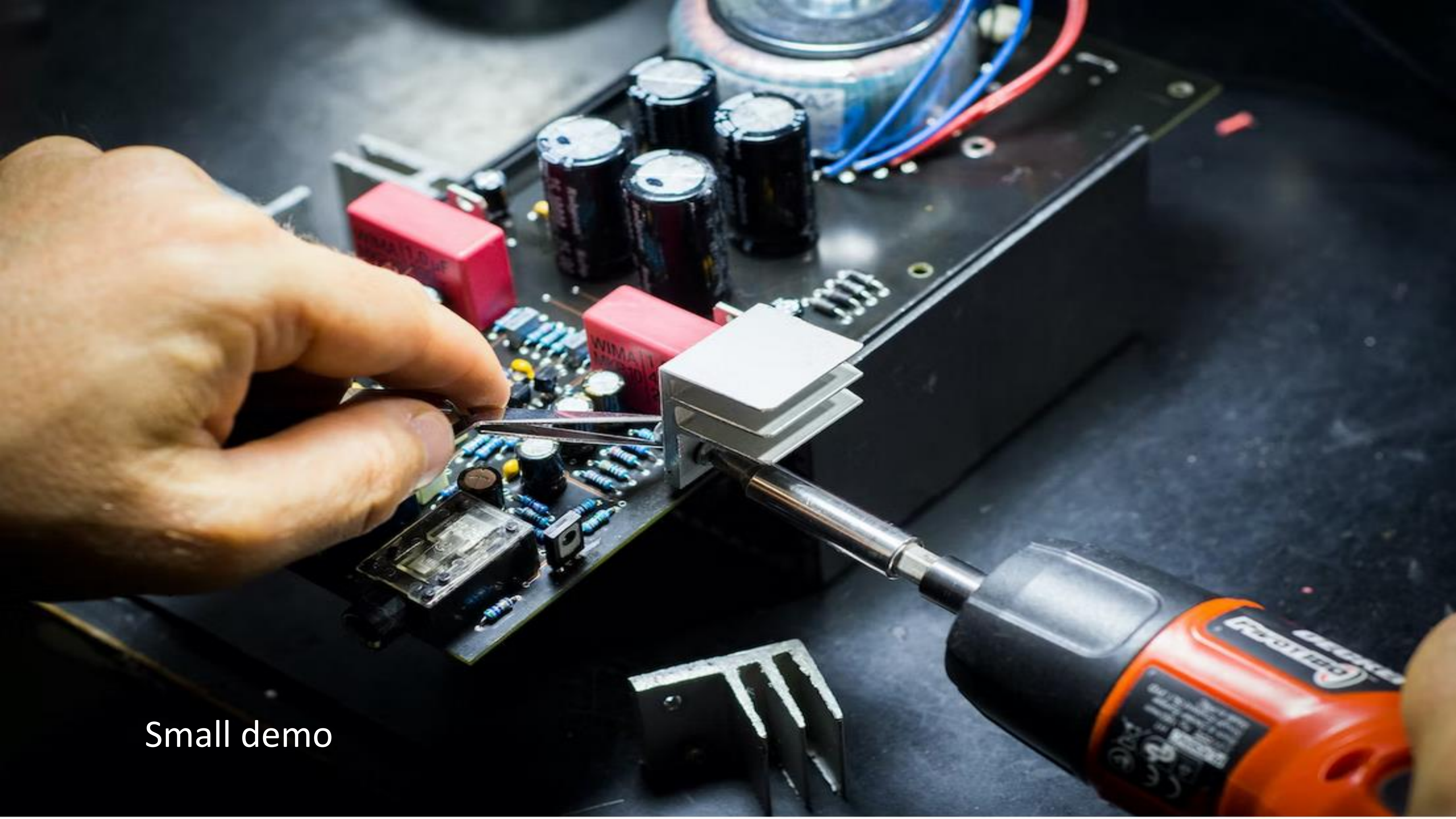


The new setup - take two

Next approach

What further was discussed

- Can we setup logical replication based on a backup?
  - You can't
  - You can only restore into a new managed service using those backups
- Can we create a basebackup from that managed instance and start from there?
  - Again, you cannot setup logical replication based on a backup
  - In a public cloud you cannot even use pg_basebackup
    - You don't have super user permissions
- Can we setup logical replication based on dump?
  - Can you?



Small demo

PostgreSQL logical replication

Setup logical replication based on a dump

The following is one little shell script, explained step by step

- What it does
  - Initialize a small pgbench schema in the source
  - Create the same schema, without data, in the target
  - Create a publication for three out of the four tables in the source
  - Create a subscription for the three tables in the target
  - Verify logical replication is fine
  - Create a publication for the fourth table in the source
  - Create a replication connection to the source database and create a snapshot
  - Dump the data of the fourth table from the snaphot
  - Load into the target
  - Create a subscription for the fourth table starting at the snapshot created above
  - Verify that logical replication is working fine

PostgreSQL logical replication

Setup logical replication based on a dump

The script, explained

```
#!/bin/bash

# These are the ports of the source and the target instance
SRCPORT=8888
TGTPORT=8889

# Cleanup in case you want to re-run the demo
psql -p 8888 -c "drop publication pub_test";
psql -p 8888 -c "drop publication pub_test_2";
psql -p 8888 -c "drop table
pgbench_accounts,pgbench_branches,pgbench_history,pgbench_tellers"
psql -p 8889 -c "drop subscription sub_test";
psql -p 8889 -c "drop subscription sub_test_2";
psql -p 8889 -c "drop table
pgbench_accounts,pgbench_branches,pgbench_history,pgbench_tellers"
```

PostgreSQL logical replication

Setup logical replication based on a dump

The script, explained

```
# intialize some demo data
pgbench -p ${SRCPORT} -i -s 10
psql -p ${SRCPORT} -c "\d"
# create one publication for the smaller tables
psql -p ${SRCPORT} -c "create publication pub_test for table
                       pgbench_branches,pgbench_history,pgbench_tellers;"

# create the empty schema in the target
pg_dump -p ${SRCPORT} --schema=public --schema-only | psql -p ${TGTPORT}

# create the first subscription for the three tables
psql -p ${TGTPORT} -c "create subscription sub_test connection 'host=localhost
port=${SRCPORT} user=postgres dbname=postgres' publication pub_test;"

# Get the meta data of the subscription
psql -p ${TGTPORT} -c "select * from pg_subscription;"
```

PostgreSQL logical replication

Setup logical replication based on a dump

The script, explained

```
# Verify that data has been loaded
psql -p ${TGTPORT} -c "select count(*) from pgbench_branches;"
psql -p ${TGTPORT} -c "select count(*) from pgbench_branches;"

# Verify the replication is ongoing
psql -p ${SRCPORT} -c "insert into pgbench_branches values (-1,-1,'aa');"
psql -p ${TGTPORT} -c "select * from pgbench_branches where bid = -1;"

# Create the second publication for the "large" table
psql -p ${SRCPORT} -c "create publication pub_test_2 for table pgbench_accounts;"
psql -p ${SRCPORT} -c "select * from pg_publication;"

# create a snapshot to dump from
# This is a replication connection and must be kept open,
# so you need a new session from here on
psql -p ${SRCPORT} "dbname=postgres port=${SRCPORT} replication=database"
CREATE_REPLICATION_SLOT my_logical_repl_slot LOGICAL pgoutput;
```

PostgreSQL logical replication

Setup logical replication based on a dump

The script, explained

```
# Dump from the snapshot (of course you need to adjust the snapshot ID)
pg_dump -p ${SRCPORT} --snapshot=00000004-00000020-1 -a -t public.pgbench_accounts >
pgbench_accounts.sql
# Load & verify the data
psql -p ${TGTPORT} -f pgbench_accounts.sql
psql -p ${TGTPORT} -c "select count(*) from public.pgbench_accounts;"

# create the subscription against the slot from above
psql -p ${TGTPORT} -c "create subscription sub_test_2 connection 'host=localhost
port=${SRCPORT} user=postgres dbname=postgres' publication pub_test_2 with ( slot_name =
'my_logical_repl_slot', create_slot='false' , enabled='false', copy_data='false');"
# Start the replication
psql -p ${TGTPORT} -c "alter subscription sub_test_2 enable;"
```

PostgreSQL logical replication

Setup logical replication based on a dump

The script, explained

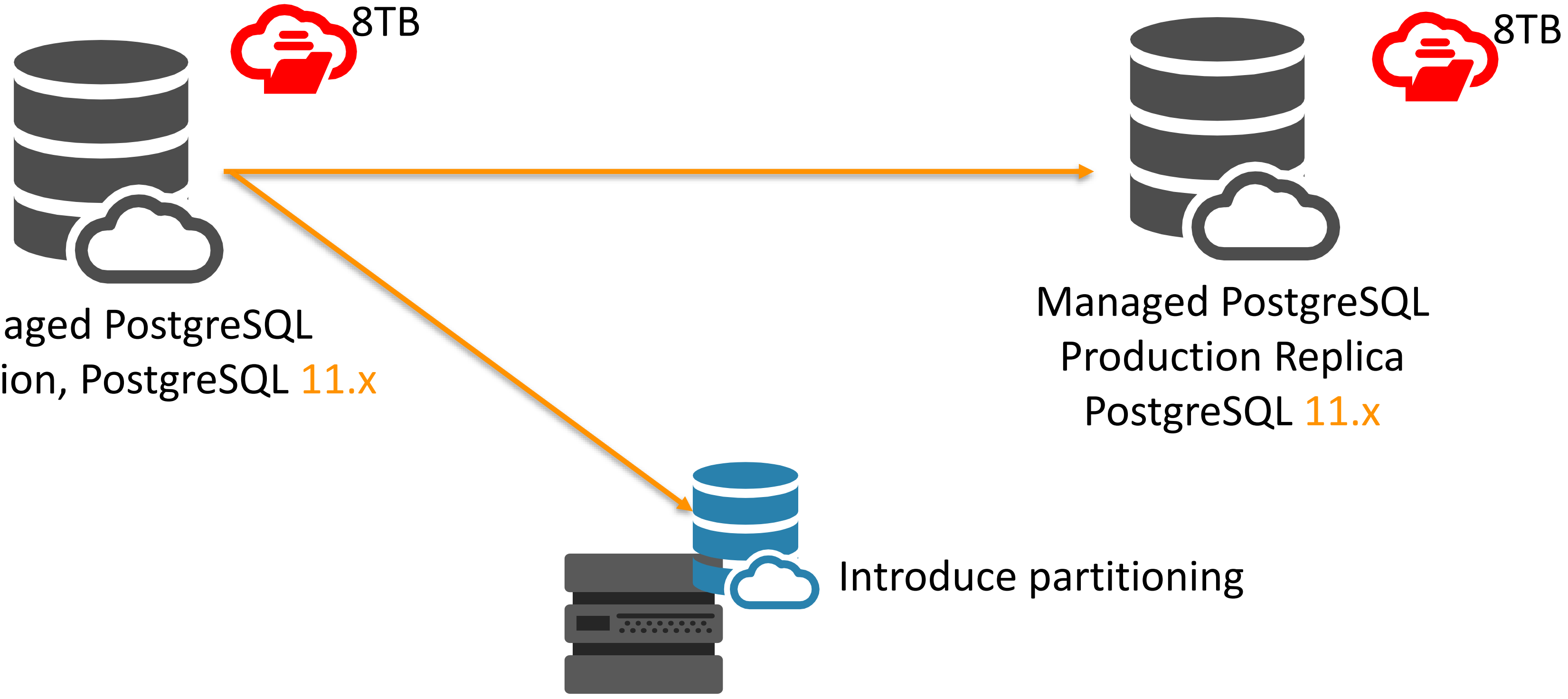
```
# Verify ongoing replication
psql -p ${SRCPORT} -c "insert into pgbench_accounts select i,i,i,i::text from
generate_series(1000001,1000100) i;"
psql -p ${TGTPORT} -c "select count(*) from public.pgbench_accounts ;"

# Exit from the replication connection
\q
```

What we finally had to do

The final setup

What we had to do

The final setup was still this, but …



8TB

8TB

Managed PostgreSQL
Production, PostgreSQL 11.x

Managed PostgreSQL
Production Replica
PostgreSQL 11.x

Introduce partitioning

Self-Managed PostgreSQL
Logical replication target
PostgreSQL 15.x

The final setup
What we had to do

The final setup was still this, but ...

- Instead of using only a few publication and subscriptions
  - Seperate the setup of logical replication into smaller pieces
    - Small schemas got their own publications and subscriptions
    - Larger schemas were broken up
      - This is easy if there are no foreign keys
      - When there are, put related tables in a separate publication / subcription
      - The three largest tables got their own publication / subcription
  - Downside?
    - Creating more pulications requires?
      - Increasing max_replication_slots, which requires?
      - A restart of production
- Sequences need to be replicated manually at the time of the switch

The final setup

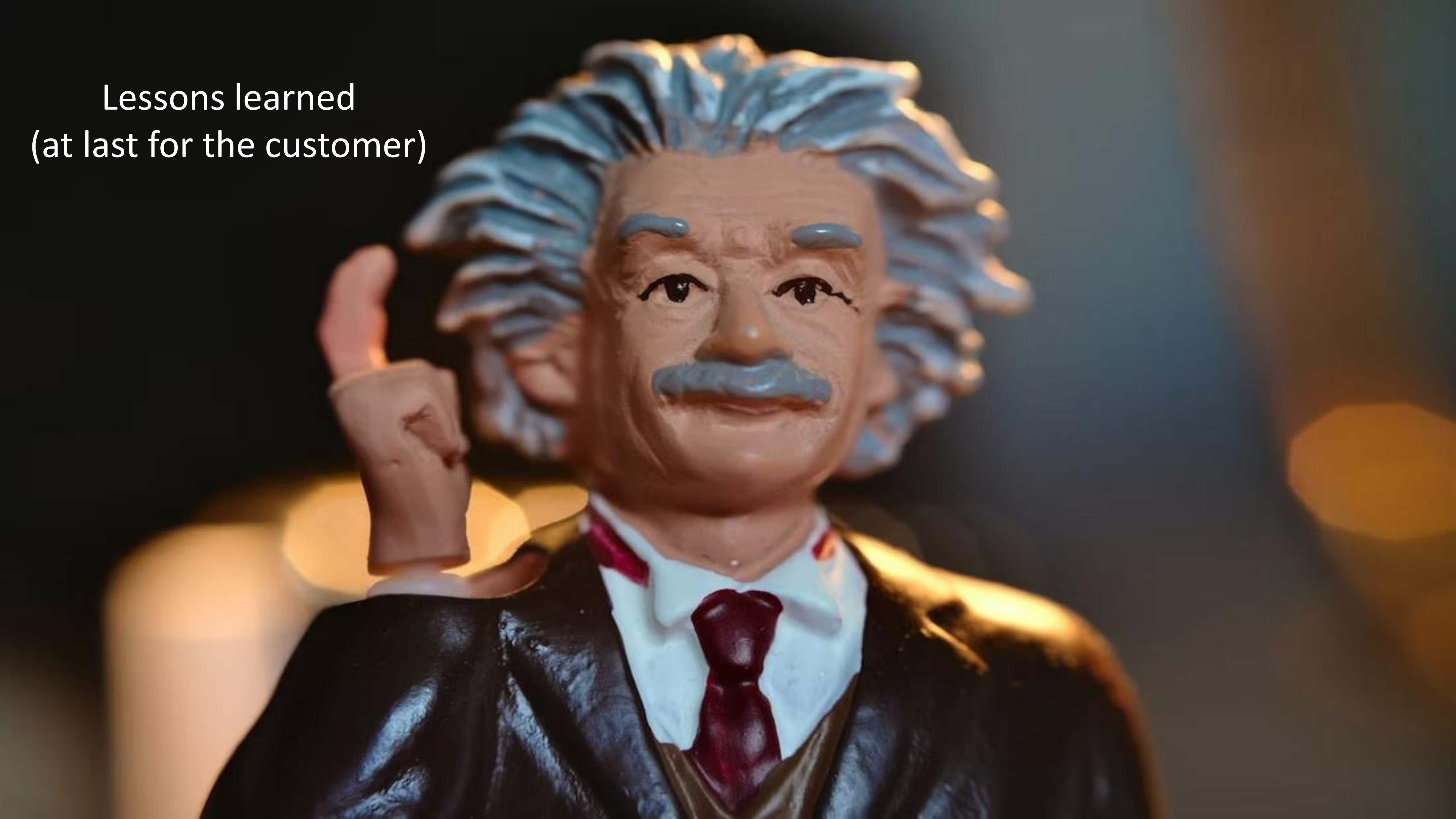
What we had to do

Other reasons for the self managed target setup

- We have a real superuser
- The next step (if required) becomes much easier
  - Going back on-prem

What options do we have now?

- Once more using logical replication, or
- Create a physical replica on-prem and let it catch up
  - We can now use pg_basebackup
  - This will usually introduce costs for outgoing network traffic



Lessons learned
(at last for the customer)

Lessons learned

When you decide to go for a managed service in a public cloud

- Make yourself familiar with the costs
  - There are costs for storage
    - Don't forget the storage costs for backups
  - There are costs for compute
  - There might be costs for network traffic
  - The faster you want to go, the more costs you will generate
- Make youself familiar with the limitations
  - No superuser
  - What extensions do you need?
  - What are the possibilities when it comes to monitoring?
- Think about how you can escape such a service in advance
  - Once you need to, the strategy should be there
  - ... and the strategy must have been tested

Any questions?

Please do ask!

We would love to boost your IT-Infrastructure

How about you?