Pythian

Migrating 100+ Postgres databases to the Cloud

Nelson Calero
Principal Consultant
Pythian

Luke Davies
Principal Consultant
Pythian

PGConf.EU 2025
October 21-24, Riga, Latvia

Agenda

1. Project description
2. Migration options
3. How we did it
4. Problems solved
5. Lessons learned
6. Appendix - for reference

Nelson Calero

Pythian - Principal Consultant

- 25+ years Database experience
- Oracle ACE Director
- Community Volunteer - LAOUC / UYOUG leader
- Montevideo, Uruguay

LinkedIn: https://www.linkedin.com/in/ncalero
Twitter / X: @ncalerouy

Luke Davies

Pythian - Principal Consultant

- 30+ years Database experience
- Chelmsford, UK

LinkedIn: https://www.linkedin.com/in/lukedavies

27

Years in Business

400+

Experts across every Data Domain & Technology

350+

Global Customers

The project

Problem description – initial information

- PostgreSQL 9.6 instances running on VMs
- 10 servers, 100+ databases
- Several multi-TB databases
- Using inheritance, PostGIS, and special data types
- Looking to migrate to a Cloud managed services with minimal downtime (GCP)
- Already evaluated replication tools and found problems: Google DMS, Striims
- Many tables not having PK

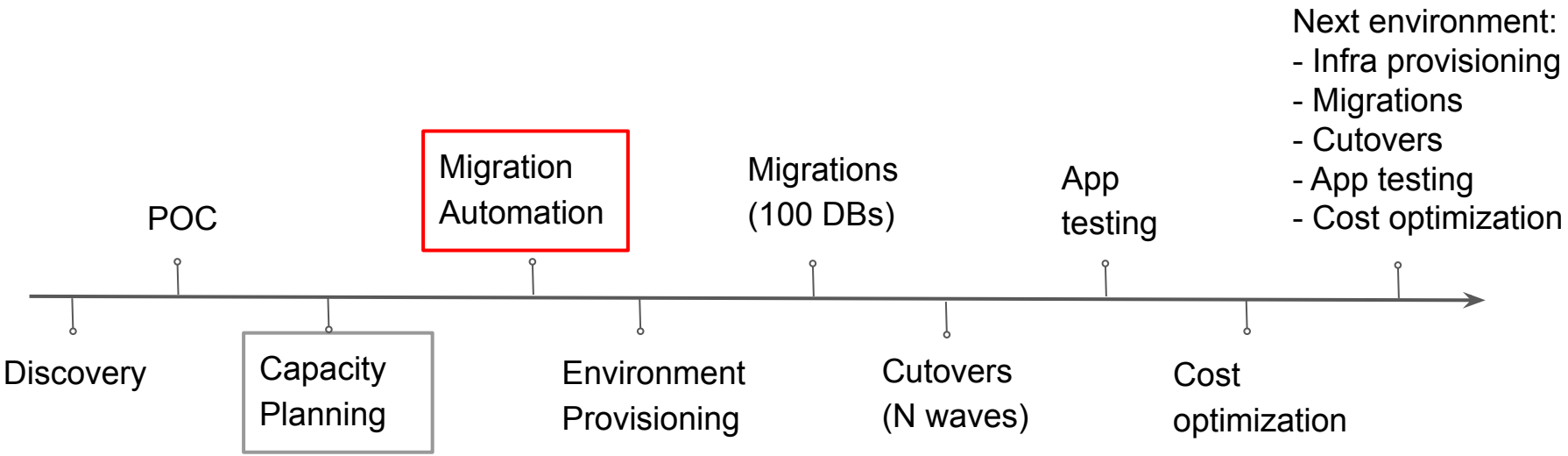
Problem description – initial information + discovered

- PostgreSQL 9.6 instances running on VMs
- 10 servers, 10
- Several multi-T
- Using inheritar
- Looking to mig
- Already evalua
- Many tables nc

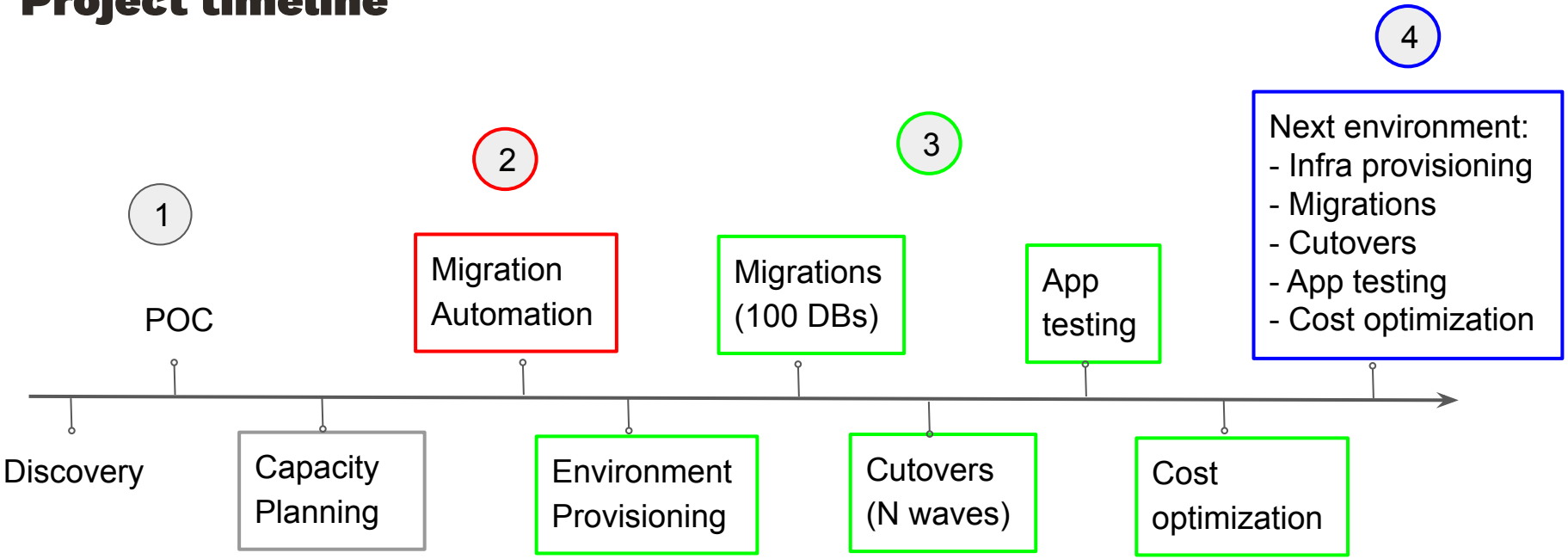
Discovered a lot of customizations

- Problems with replication tools tested were not well documented
- Test and Dev environments does not have the same data sizes as production
- Application to database mappings is not well documented
- 3 months deadline to complete the migration (platform and financial constraints)
- Tables without PK are insert only
- Source and target DBs on different networks VPCs, no direct connectivity
- Use private IPs for DBs - apps will connect using Private Service Connect (PSC)
- Looking to implement some changes in the target, as part of the migration
  - Use GCP Identity and Access Management (IAM) authentication
  - Regroup databases to align with application usage
  - Move from inheritance to declarative partitioning
  - Not all schemas will be migrated
  - Not all data from large tables is needed (some will be archived)

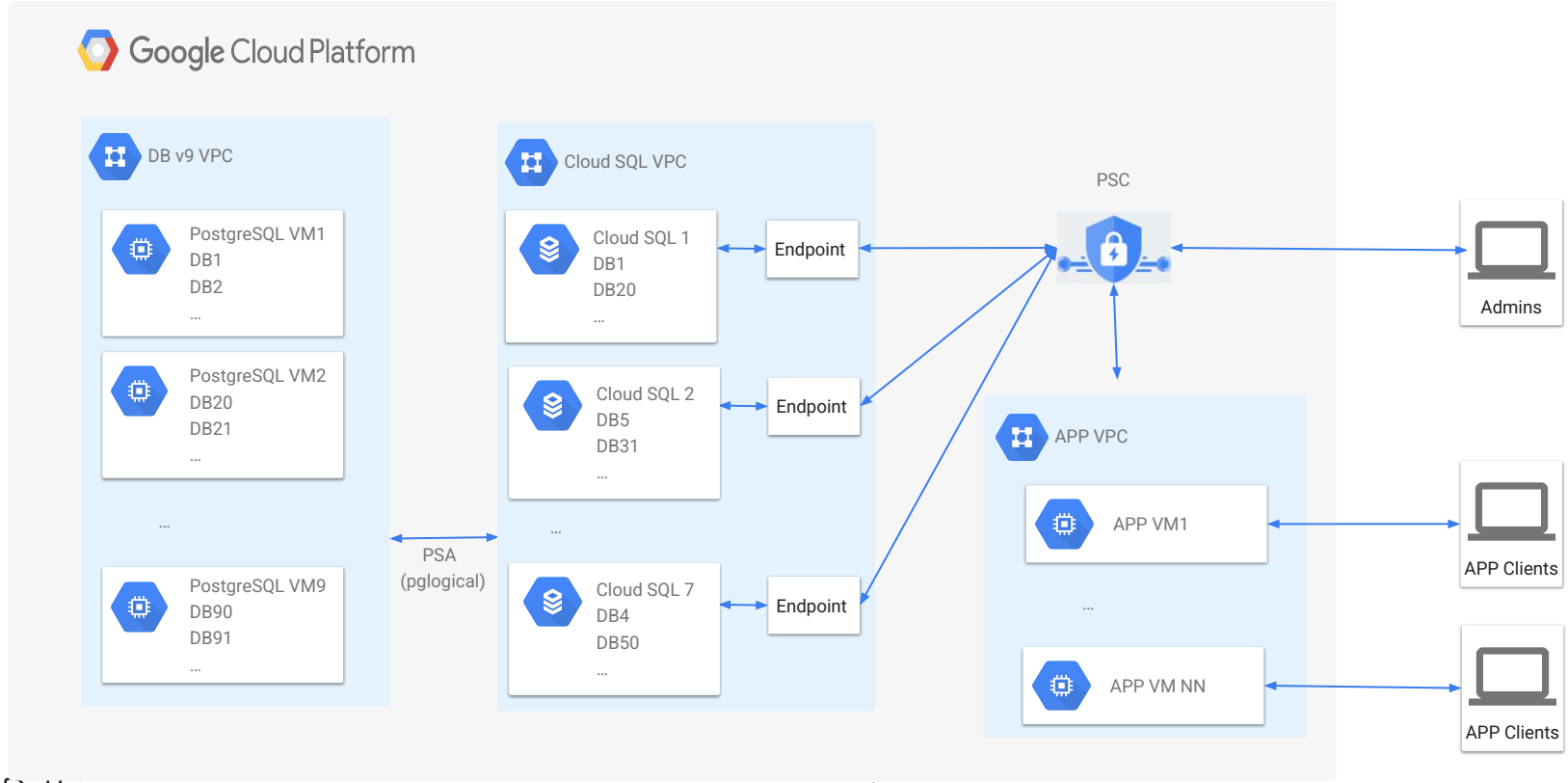
Project timeline



Project timeline



Target platform (including old DBs for migration)

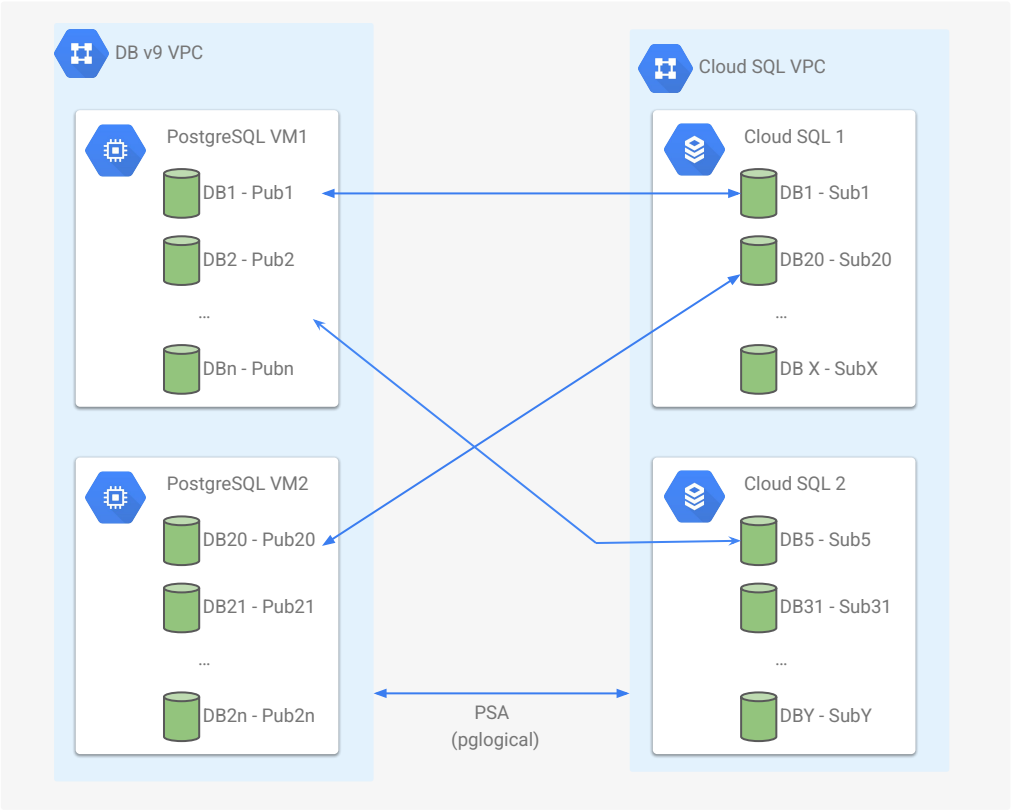


Migration options

Migration options explored

- Google DMS - https://cloud.google.com/database-migration
  - No extra cost for homogenous migrations to Cloud SQL
  - Features improved since the customer's POC, but only possible to migrate all DBs in an instance at that time (July/2024)
  - No documented problems with inheritance, PostGIS or data types
  - No time to evaluate with all DBs to decide (based on customer past experience)
- Third-party replication tools
  - License cost implications
  - No documented problems with inheritance, PostGIS or data types
  - Customer aversion given the experience with one of them, and limited time to test them
- Pglogical extension - https://github.com/2ndQuadrant/pglogical
  - Simple POCs didn't show any problems
  - Need to develop automation to use it at scale

Migration options explored - pglogical extension



How

- One Publisher per source DB
- One Subscriber per target DB
- Connection initiated by target
- Mix of data flows between Clusters

Setup

- PSA enabled on Cloud SQL VPC
- pg_hba.conf changes on source
- PostgreSQL parameters changes in both source and target for pglogical

How we did it

Project decisions

- Initial discovery to document all source DBs complexities
  - version, extensions, data types, sizes
- POC of the entire migration using pglogical for three representative DBs
  - size, features (PostGis, inheritance, data types), and app complexity
- Capacity planning to size the target clusters (and validate the initial proposed mapping)
- Automate as much as possible to minimize manual interventions and errors
  - Terraform to provision - VPC, IAM, Cloud SQL instances, roles and IAM attachments
  - A migration framework using open source tools - not reinventing the wheel
    - Used for databases creation and data migration
  - Cutover scripts - manual intervention was required in coordination with app team
- No big-bang migration, to reduce application risks
  - several migration waves of ~20 DB each, in a two hours migration window

Project decisions

- Initial discovery to document all source DBs complexities
  - version, extensions, data types, sizes
- **POC of the entire migration using pglogical for three representative DBs**
  - size, features (PostGis, inheritance, data types), and app complexity
- **Capacity planning to size the target clusters (and validate the initial proposed mapping)**
- Automate as much as possible to minimize manual interventions and errors
  - Terraform to provision - VPC, IAM, Cloud SQL instances, roles and IAM attachments
  - **A migration framework using open source tools - not reinventing the wheel**
    - Used for databases creation and data migration
  - **Cutover scripts - manual intervention was required in coordination with app team**
- No big-bang migration, to reduce application risks
  - several migration waves of ~20 DB each, in a two hours migration window

pglogical extension POC

Few issues found

- Slow initial data copy for large tables (+20 hours for a 80Gb table)
- Even slower when having several indexes, or large toast
- Ownership and grants errors
  - No superuser allowed in GCP - *cloudsqlsuperuser* role instead (but not the same) https://cloud.google.com/sql/docs/postgres/users

Solutions:

- Not a problem if WAL file retained on source (per replication slot) is not big
  - It was our case, even with heavy concurrency we saw few GB in 24 hours
- Speeding up subscription initial data copy:
  - Before starting subscriptions, drop non PK indexes on large tables
  - Recreate indexes after data copy - parameters tweaked (parallelism & memory)
  - GIST indexes took hours for large tables (+100GB), no parallel and/or online options
  - Initial data copy + indexes creation reduced to 8 hours in total from 32 hours.
- Tested dump/restore for the initial copy - complex and not safe with concurrent sessions

Capacity planning

Extracting PostgreSQL metrics per database

- Evaluated pgcluu - https://github.com/darold/pgcluu
  - Simple and nice reports to explore metrics captured, but grouped by cluster
- Implemented crontab script gathering stats per database
  - Hourly snapshots from **pg_stats_database**
  - Minutely snapshots of session state from **pg_stats_activity**
- **track_io_timing** parameter was disabled in all production clusters (default)
  - If enabled, block_read/write_time stats are captured
  - performance overhead (not tested, as per docs)

Analysis of metrics captured

- Imported data into a local PostgreSQL DB - *db_stats* and *connection_stats* tables
- Created view to get delta values for stats - *db_stats_snap_v*
- Created table with DB mappings to explore target instances - *sql_instances*
- Created an XLS and charts using pivot tables (mainly commits, reads and hits delta values)

Migration automation

Design decisions:

- Open source tools - bash and YAML configuration files
- Parallel database migrations
- Resume capabilities
- Summary and detailed Logging
- Cutover process in waves

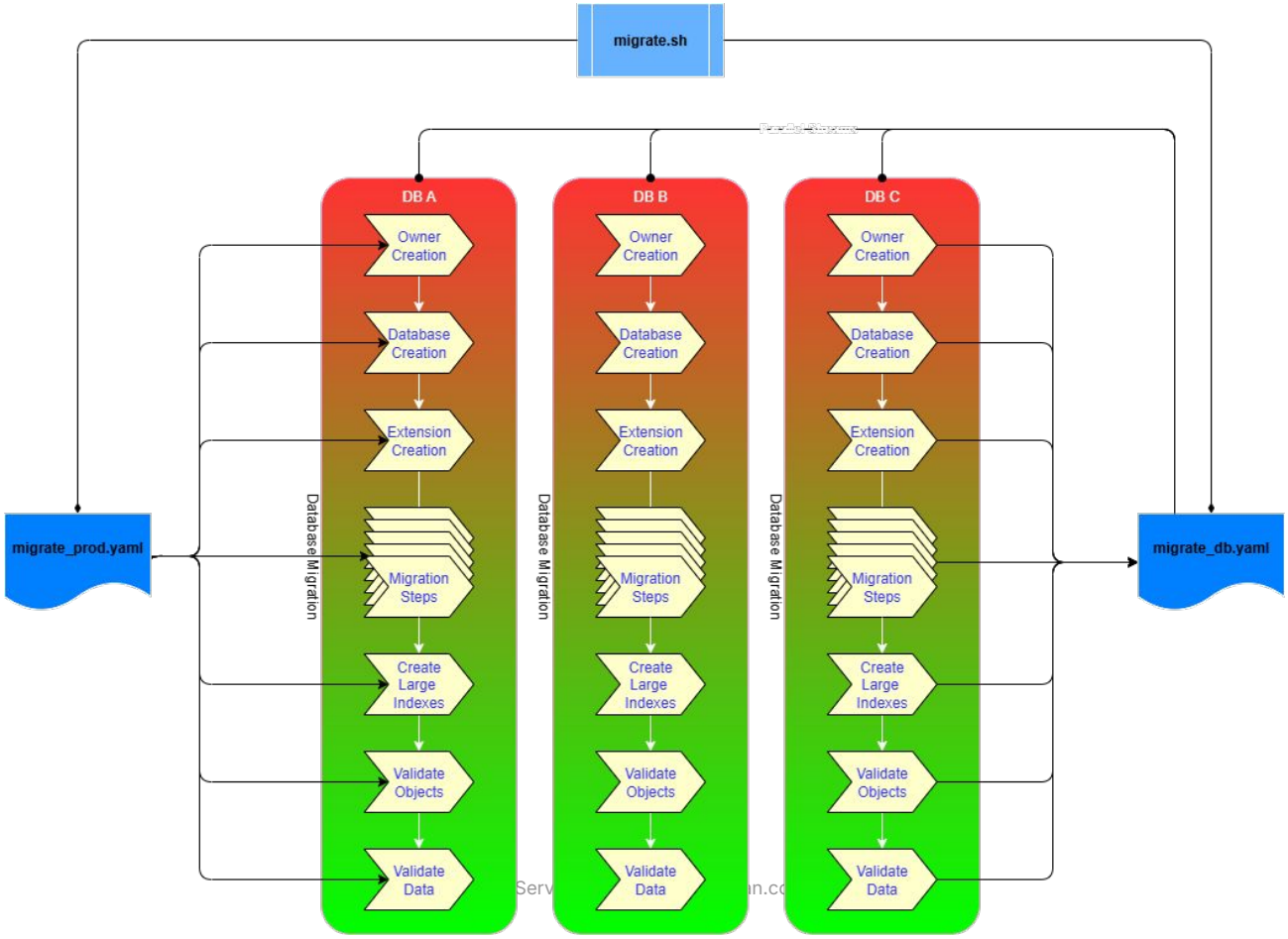
Configuration files:

- YAML file for migration steps
- YAML file for databases to process, including relevant content
    - source and target instances, schemas, data to discard, wave group, etc.

Main migration script in bash

- Reading configuration and calling scripts needed on each step

Migration automation - Overview



Migration automation - Relevant steps

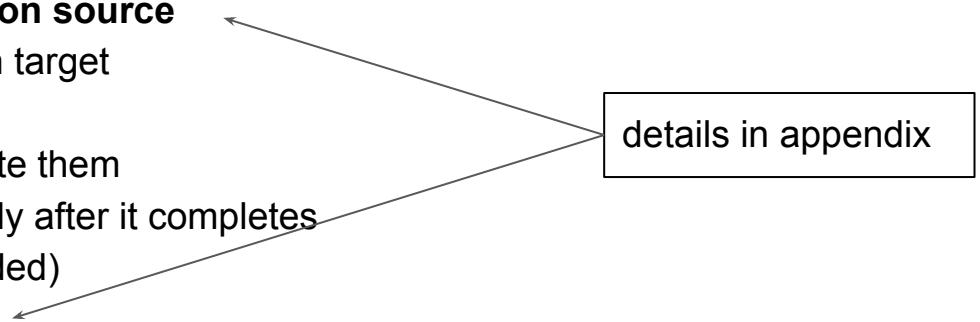
Several steps involved, pglogical being one of them

- Target instance creation (terraform)
- Target IAM users attachment (gcloud commands)
- Schema creation (pg_dump/restore)
  - Ownerships and grants adjustment (sed on the export file)
- Pglogical installation on source and target
- Pglogical publisher configuration on source
- Pglogical subscriber configuration on target
  - Drop indexes for large tables
  - Generate statements to recreate them
- Monitor initial data copy, continue only after it completes
- Execute index creation script (if needed)
- Schema and data verification
  - Count rows for small tables
  - Hash over all columns for a sample rows - all tables
- Vacuum Analyze

Migration automation - Relevant steps

Several steps involved, pglogical being one of them

- Target instance creation (terraform)
- Target IAM users attachment (gcloud commands)
- Schema creation (pg_dump/restore)
  - Ownerships and grants adjustment (sed on the export file)
- Pglogical installation on source and target
- **Pglogical publisher configuration on source**
- Pglogical subscriber configuration on target
  - Drop indexes for large tables
  - Generate statements to recreate them
- Monitor initial data copy, continue only after it completes
- Execute index creation script (if needed)
- **Schema and data verification**
  - Count rows for small tables
  - Hash over all columns for a sample rows - all tables
- Vacuum Analyze



Migration automation – Sample execution

```
$ bin/migrate.sh -v -c migrate_prod -t db-target2 -d db5
MAIN: Configuration file set to /migra/config/migrate_prod.yaml
MAIN: Database Target Server set to db-source1
MAIN: Database Name set to db5
MAIN: Getting servers ...
MAIN: Processing Server db-source1
MAIN: Getting databases ...
MAIN: Processing Server db-source3
MAIN: Getting databases ...
MAIN: Processing Database db-source3 -> db5
SetGlobals: ReplicaUser set to app_replica
SetGlobals: PostgresUser set to ncalero
SetGlobals: TotalParallel set to 4
SetGlobals: URLBase set to prod.mydb.com
SetGlobals: SuperUser set to dba
SetGlobals: OwnerUser set to db_owner
SetGlobals: Variable ScriptBaseDir unset. Defaulting to /migra
SetGlobals: ScriptBaseDir set to /migra
SetGlobals: Variable ScriptBinDir unset. Defaulting to /migra/bin
SetGlobals: ScriptBinDir set to /migra/bin
SetGlobals: Variable ScriptSQLDir unset. Defaulting to /migra/sql
SetGlobals: ScriptSQLDir set to /migra/sql
SetGlobals: Variable ScriptSQLgenDir unset. Defaulting to /migra/gensql
SetGlobals: ScriptSQLgenDir set to /migra/gensql
MAIN: Run Migrate job for database db-source3 -> db5 set off with PID 1878913
RunMigrate: Migrating database db-source3 -> db5 to target db-target2 using pglogical
...
```

Migration automation – Sample execution

```
$ bin/migrate.sh -v -c migrate_prod -t db-target2 -d db5
MAIN: Configuration file set to /migra/config/migrate_prod.yaml
MAIN: Database
MAIN: Database
MAIN: Getting
MAIN: Processi
MAIN: Getting
MAIN: Processi
MAIN: Getting
MAIN: Processi
SetGlobals: Re
SetGlobals: Pc
SetGlobals: To
SetGlobals: UR
SetGlobals: Su
SetGlobals: Ow
SetGlobals: Va
SetGlobals: ScriptBaseDir set to /migra
SetGlobals: Variable ScriptBinDir unset. Defaulting to /migra/bin
SetGlobals: ScriptBinDir set to /migra/bin
SetGlobals: Variable ScriptSQLDir unset. Defaulting to /migra/sql
SetGlobals: ScriptSQLDir set to /migra/sql
SetGlobals: Variable ScriptSQLgenDir unset. Defaulting to /migra/gensql
SetGlobals: ScriptSQLgenDir set to /migra/gensql
MAIN: Run Migrate job for database db-source3 -> db5 set off with PID 1878913
RunMigrate: Migrating database db-source3 -> db5 to target db-target2 using pglogical
...
```

```
...
RunMigrate: Migrating database db-source3 -> db5 to target db-target2 using pglogical
RunMigrate: Database db-source3 -> db5 starting from stage 60
MAIN: Processing Server db-source3
MAIN: Getting databases ...
RunStep: Running migration stage 70 for Database db-source3 -> db5
MAIN: Processing Server db-source4
MAIN: Getting databases ...
MAIN: Processing Server db-source5
MAIN: Getting databases ...
RunStep: Database db-source3 -> db5 running stage 70 -> Fix up Database Object MetaData Dump
RunStep: Database db-source3 -> db5 running script -> /migra/bin/fix_pgdump.sh -l
/migra/logs/migrate_db5_db5_250709173710.log -h db-source3.prod.mydb.com -d db5 -s 10.100.0.11 db_owner
...
```

Migration automation – Sample execution

```
$ bin/migrate.sh -v -c migrate_prod -t db-target2 -d db5
MAIN: Configuration file set to /migra/config/migrate_prod.yaml
MAIN: Database
MAIN: Database
MAIN: Getting
MAIN: Processi
MAIN: Getting
...
RunMigrate: Migrating database db-source3 -> db5 to target db-target2 using pglogical
RunMigrate: Database db-source3 -> db5 starting from stage 60
MAIN: Processing Server db-source3
```

```
$ bin/status.sh -w w4
Filters:  Target=   Wave=w4
General info                                          Replica (target)    Publisher (source)
Database  Size GB     Method Stage  Wave  Completed Con/Run  Rep Status Con/Run     Pub Lag      Source     Target
--------  -------     ------ -----  ----  --------- -------  ---------- -------     -------  ----------- -----------
db1        312.00  pglogical   115    w4       true     2/2        good     2/2        good  db-source1  db-target1
db67      1102.00  pglogical   105    w4      false     2/1         bad     1/1        good  db-source1  db-target1
db5       3464.00  pglogical   105    w4      false     2/0     Copying     2/0         bad  db-source3  db-target2
db11         7.02  pglogical   115    w4       true     1/1        good     1/1 good/Lag(1)  db-source2  db-target1
db24        95.00  pglogical   115    w4       true     2/2        good     2/2        good  db-source2  db-target3
db32       234.00  pglogical   115    w4       true     2/2        good     2/2 good/Lag(1)  db-source2  db-target3
db18       278.00  pglogical   115    w4       true     2/2        good     2/2 good/Lag(2)  db-source4  db-target3
db41       271.00  pglogical   115    w4       true     2/2        good     2/2        good  db-source4  db-target3
db89         1.84  pglogical   115    w4       true     2/2        good     2/2        good  db-source4  db-target4
db52         0.01  pglogical   115    w4       true     1/1        good     1/1        good  db-source5  db-target4
db9          5.18  pglogical   115    w4       true     2/2        good     2/2 good/Lag(2)  db-source5  db-target1
-----     -------    -------              --------- -------     ------- -------
Total     5770.05         11                   9   20/17           9   19/17
```

```
...
```

Migration automation – Cutover

- Pre-cutover validations (shell script - day before cutover)
  - Replication status (primary and replica)
  - Replication slots details (primary)
  - Publishers and subscribers' details
  - WAL directory files and size in primary
  - Table Statistics update and status (using analyze)
  - Accounts attached to the target server(s)
  - Owners of database objects
- Cutover steps
  - Show database connections (source and target)
  - Show RO and RW role grants for SA accounts (target)
  - Application switchover
  - Update sequence values to match source values (target)
  - Grant RW roles to SA accounts (target)
- Post-cutover validations (days after cutover)
  - Show database connections (source and target)
  - Remove replication configuration (subscribers and publishers)

Problems faced (and solved)

Problems solved

- pg_dump: Reusability
- pg_dump: Compatibility
- pglogical Replication: Storage Space Errors
- pglogical Replication: Temporary Space Errors
- pglogical Replication: Subscription worker remains down
- pglogical Replication: Non-partitioned to partitioned table
- pg_dump: Inheritance column order (dual parent)
- pg_dump: Inheritance column order (object creation time)
- postGIS: Raster
- Postgres v14: Invalid Index (limiting the column size)
- Postgres v9.6: Count bug
- Python psycopg2 module: far future dates
- Validation: Floating point differences
- Validation: Finding functions in v9.6 vs v14
- GCP IAM: audit user column size

Problems solved - pg_dump: Reusability

Problem

The extract file from pg_dump produces SQL but if this SQL runs more than once it fails with duplicate names etc.

Solution

- Add **`IF NOT EXISTS`** clauses (`TABLES`, `SEQUENCES`, `INDEXES`, `MVIEWS`)
- Add **`OR REPLACE`** clauses (`FUNCTIONS`, `VIEWS`, `TRIGGERS`)
- Add pgPL/SQL wrappers to objects (`DOMAINS`, `CONSTRAINTS`)
- Remove superuser only objects (`OPERATOR CLASS/FAMILY`)
- Privileges must be split out to evaluate individually
  - `GRANT SELECT,INSERT ON TABLE …`
- Some objects not needed (`CREATE SCHEMA`, `ALTER DEFAULT PRIVILEGES`)
- Used **`sed`** and **`awk`** to edit the files

Problems solved - pg_dump: Compatibility

Problem

Some object types were deprecated between versions 9.6 and 14

Solution

- Edit the file using **sed**
  - `anyarray ⇒ anycompatiblearray`
  - `anyelement ⇒ anycompatible`

Problems solved - pglogical Replication: Storage Space Errors

Problem

Received errors on Cloud SQL console when storage had run out and it was auto extending

```
2025-05-20 11:30:42.733 UTC [26496]: [1-1] db=v,user=app_replica
ERROR:  could not extend file "base/31213/32985": No space left on device

2025-05-20 11:30:53.005 UTC [26176]: [3-1] db=i,user=[unknown] DETAIL:  destination connection reported:
ERROR:  could not extend file "base/21381/26093.9": No space left on device
```

Solution

Create instance with the expected final size.

Note: the maximum storage size was big enough to increase when we got those errors.
This happened because the auto-extension didn't happen as fast as the import process was moving.

Problems solved - pglogical Replication: Temporary Space Errors

Problem

- When creating large indexes on the target (migration step to optimize initial data copy)
- Also when using filters on the publisher (using pglogical, per table), when the subscription executes the initial data copy:

```
ERROR:  temporary file size exceeds temp_file_limit (1021877kB)
```

Solution

- Change the **`temp_file_limit`** to 25G (Default 10% of initial disk size) - online operation
- For pglogical errors with filtered tables, needs to be changed on the source cluster

Note: on the target, this problem is more relevant when using autoextensible disk, as this value can get a small size compared with the desired final disk size.

https://cloud.google.com/sql/docs/postgres/flags

Problems solved - pglogical Replication: Subscription worker remains down

Problem

The subscriber remains down and we get the following log entry

```
2025-05-20 13:48:04.809 UTC [39716]: [2-1] db=pp,user=[unknown]
ERROR:  worker registration failed, you might want to increase max_worker_processes setting
```

Solution

Increase **`max_worker_processes`** - requires instance reboot

NOTE: May need to increase instance memory footprint to allow for increased `max_worker_processes`

Problems solved - pglogical Replication: Non-partitioned to partitioned table

Problem

When trying to replicate a “normal” table to a partitioned table, the target cluster crashed and then entered a crash loop.

Solution

This operation cannot be done if the source is PostgreSQL 10 or older.
Reference: https://www.enterprisedb.com/docs/pgd/3.7/pglogical/replication-sets/

To recover the instance:

- **Disable pglogical extension** on the Cloud SQL console
- Bring up the cluster
- **Drop the subscription** on the offending database.
- Re-enable the pglogical extension
- Restart the cluster

Problems solved - pg_dump: Inheritance column order (dual parent)

Problem

A dual parent inherited table can be created with a different column order if the parents are not ordered correctly

```
CREATE TABLE sales.phone (
        name varchar NOT NULL,
        from_number varchar NOT NULL,
        CONSTRAINT phone_pkey PRIMARY KEY (phone_call_id)
)
INHERITS (sales.phone_call, sales.phone_call_log);
```

Solution

Switch the order of the parents

```
CREATE TABLE sales.phone (
        name varchar NOT NULL,
        from_number varchar NOT NULL,
        CONSTRAINT phone_pkey PRIMARY KEY (phone_call_id)
)
INHERITS (sales.phone_call_log, sales.phone_call);
```


Problems solved - pg_dump: Inheritance column order (object creation time)

Problem

In some cases where there is inheritance the column order is dependent on object creation time

1. Parent table (columns ⇒ p1, p2)
2. Child table 1 inherited from parent with extra column (columns ⇒ p1, p2, c1_col)
3. Add column to parent table (columns ⇒ p1, p2, p3)
4. Child table 1 (columns ⇒ p1, p2, c1_col, p3)
5. A new child table 2 is created, inheriting from parent when it already had the extra column:
   (columns ⇒ p1, p2, p3, c2_col)

On recreation (when using the pg_dump generated file) get the following:

1. Parent table (columns ⇒ p1, p2, p3) ✔
2. Child table 1 (columns ⇒ p1, p2, p3, c1_col) ✘
3. Child table 2 (columns ⇒ p1, p2, p3, c2_col) ✔

Problems solved - pg_dump: Inheritance column order (object creation time)

Solution

After table is restored (no data)

`parent(p1,p2,p3)` ✔ ; `child1(p1,p2,p3,c1_col)` ✘ ; `child2(p1,p2,p3,c2_col)` ✔

1. Drop parent column (p3)

   `parent(p1,p2)` ✘ ; `child1(p1,p2,c1_col)` ✘ ; `child2(p1,p2,c2_col)` ✘

2. Add parent column back again (p3) - this corrects child table 1

   `parent(p1,p2,p3)` ✔ ; `child1(p1,p2,c1_col,p3)` ✔ ; `child2(p1,p2,c2_col,p3)` ✘

3. Child table 2 is now incorrect - drop child table 2

   `parent(p1,p2,p3)` ✔ ; `child1(p1,p2,c1_col,p3)` ✔

4. Re-create child table 2 with original inheritance clause - this corrects child table 2

   `parent(p1,p2,p3)` ✔ ; `child1(p1,p2,c1_col,p3)` ✔ ; `child2(p1,p2,p3,c2_col)` ✔

Problems solved - postGIS: Raster

Problem

From postGIS V3.0 the raster functions were split into their own extension

Solution

Add extension **`postgis_raster`** when postGIS is in use

Problems solved - Postgres v14: Invalid Index (limiting the column size)

Problem

Index entries in V14 are size limited

```
pg_restore: from TOC entry 2418; 1259 17952 INDEX comp_idx db_owner
pg_restore: error: could not execute query: ERROR:  index row size 2712 exceeds btree version 4 maximum 2704 for index "comp_idx"
DETAIL:  Index row references tuple (244988,4) in relation "comp".
HINT:  Values larger than 1/3 of a buffer page cannot be indexed.
Consider a function index of an MD5 hash of the value, or use full text indexing.
Command was: CREATE INDEX "comp_idx" ON "doc"."comp" USING "btree" ("url");
```

Solution

Use a function index that hashes the value.

Note: also requires changes to the app to modify the SQL using this table to include the hash function

Example:

```
CREATE INDEX "comp_idx" ON "doc"."comp" (md5("url"));
```


Problems solved - Postgres v9.6: Count bug

Problem

There was a count problem when checking the parent table of an inheritance set of tables

- `SELECT COUNT(*) FROM p1;` ⇒ 1813806
- `SELECT * FROM p1;` ⇒ 1813796

Solution

Issuing a **`VACUUM FULL`** on the table resolves the issue

Problems solved - Python psycopg2 module: far future dates

Problem

When using python to check the data integrity found that years greater than 9999 was not handled by the module

```
ValueError: year 10222 is out of range
```

Solution

Convert the date and timestamp columns to strings prior to comparison as the database converts far future date correctly.

Problems solved - Validation: Floating point differences

Problem

When validating data that uses the **`double`** data type get decimal place issues and so get value mismatches

```
V9.6 value : 0.981682392355061
V14 value  : 0.9816823923550609
```

Solution

The initialization parameter **`extra_float_digits`** covers the precision of `double` data type output.
The default value changed in PostgreSQL v12 from 0 to 1.
Ensure that the parameter is the same on both source and target to prevent precision mismatches during validations (it does not affect stored data)

Problems solved - Validation: Finding functions in v9.6 vs v14

Problem

The SQL that finds functions changed between V9.6 and V14

V9.6

```
SELECT n.nspname||'.'||p.proname AS function_name
  FROM pg_proc p
  JOIN pg_namespace n ON n.oid = p.pronamespace
  LEFT JOIN pg_depend d ON d.objid = p.oid AND d.deptype = 'e'
 WHERE p.proisagg = false            -- not aggregate for V9.6
   AND p.proiswindow = false         -- not window function for V9.6
   AND d.objid IS NULL               -- not from extension
   AND n.nspname IN (:include_schemas)
 ORDER BY 1;
```

V14

```
SELECT n.nspname||'.'||p.proname AS function_name
  FROM pg_proc p
  JOIN pg_namespace n ON n.oid = p.pronamespace
  LEFT JOIN pg_depend d ON d.objid = p.oid AND d.deptype = 'e'
 WHERE p.prokind = 'f'               -- regular user-defined functions
   AND d.objid IS NULL               -- not from extensions
   AND n.nspname IN (:include_schemas)
 ORDER BY 1;
```


Problems solved - GCP IAM: audit user column size

Problem

Many tables had audit user columns, like created_by, modified_by, deleted_by, of varchar2(25), GCP service accounts used by applications to connect to the Cloud SQL database where larger.

```
$ gcloud sql users list --instance=db-target2
NAME                                          HOST  TYPE                          PASSWORD_POLICY
app_replica                                         BUILT_IN
dba                                                 BUILT_IN
user1@mycompany.com                                 CLOUD_IAM_GROUP_USER
user2@mycompany.com                                 CLOUD_IAM_GROUP_USER
support@mycompany.com                               CLOUD_IAM_GROUP
service-app-invoices@mycomp-prod.iam                CLOUD_IAM_SERVICE_ACCOUNT
service-api-purchases@mycomp-prod.iam               CLOUD_IAM_SERVICE_ACCOUNT
```

Solution

Modified all audit user columns length to varchar2(50).

Lessons learned

Lessons learned

- POC needs to include all the complexity to proper use it for planning
  - Initial project estimations based on the POC were short (not considering all issues discovered)
  - How big should be a buffer for unknowns?
- Customizations were discovered as the project progressed
  - Issues and data volume opened the door to new decisions - hard to anticipate
  - Flexibility to implement changes with the tools used was the key
  - Third party tools would have required a lot of interactions with the vendors to make the changes needed, if possible to request them
  - Several one-off requests that could not be automated
- Schema/objects validations were moved to earlier steps of the migration (before data transfer) to catch problems earlier and save time (specific for table column order).
- Pglogical extension is simple to use, the main migration complexity and effort is due to custom requirements, preparation, validation steps, and automating all steps until have a repeatable process.
- Main Cloud SQL cost is CPU, so maximize Instance memory use, and minimize CPU usage.

References

- Pglogical extension - https://github.com/2ndQuadrant/pglogical
- pgcluu - https://github.com/darold/pgcluu
- YQ utility: https://github.com/mikefarah/yq
- Partitioned tables and replication: https://www.enterprisedb.com/docs/pgd/3.7/pglogical/replication-sets/
- DVT: https://github.com/GoogleCloudPlatform/professional-services-data-validator
- Cloud SQL limits: https://cloud.google.com/sql/docs/postgres/quotas
- Google DMS: https://cloud.google.com/database-migration
- Cloud SQL PSA and PSC configuration: https://cloud.google.com/sql/docs/postgres/configure-private-services-access-and-private-service-connect
- Cloud SQL users: https://cloud.google.com/sql/docs/postgres/users

Questions?

calero@pythian.com
https://www.linkedin.com/in/ncalero
@ncalerouy

davies@pythian.com
https://www.linkedin.com/in/lukedavies/

Agenda

1. Project description
2. Migration options
3. How we did it
4. Problems solved
5. Lessons learned
6. **Appendix - for reference**
   - **Capacity planning**
   - **Automation config files**
   - **pglogical publisher creation**
   - **Schema and Data validation**

Appendix – Capacity planning

```
mydb=# \COPY connection_stats FROM '/tmp/all_conn_state.txt' DELIMITER ',' CSV HEADER;
COPY 816314

mydb=# \COPY db_stats FROM '/tmp/all_db_stats.txt' DELIMITER ',' CSV HEADER;
COPY 13990

mydb=# \i aas.sql
  host | snaps | cnt_sum | cnt_max | cnt_avg | act_max | act_avg | max_max_in_db
-------+-------+---------+---------+---------+---------+---------+---------------
 ins1  |   803 |     829 |       2 |   1.032 |       4 |   1.127 |             4
 inst2 |  2984 |    3953 |       5 |   1.325 |       8 |   1.366 |             5
 Inst3 |  5264 |   17539 |      12 |   3.332 |      55 |   8.256 |            28
 inst4 |  1198 |    1254 |       3 |   1.047 |       9 |   1.410 |             9
 inst5 |  5274 |   12056 |       6 |   2.286 |      29 |   9.393 |            21
 Inst6 |  3898 |    5088 |       5 |   1.305 |      32 |   3.144 |            31
...
```


Capacity planning

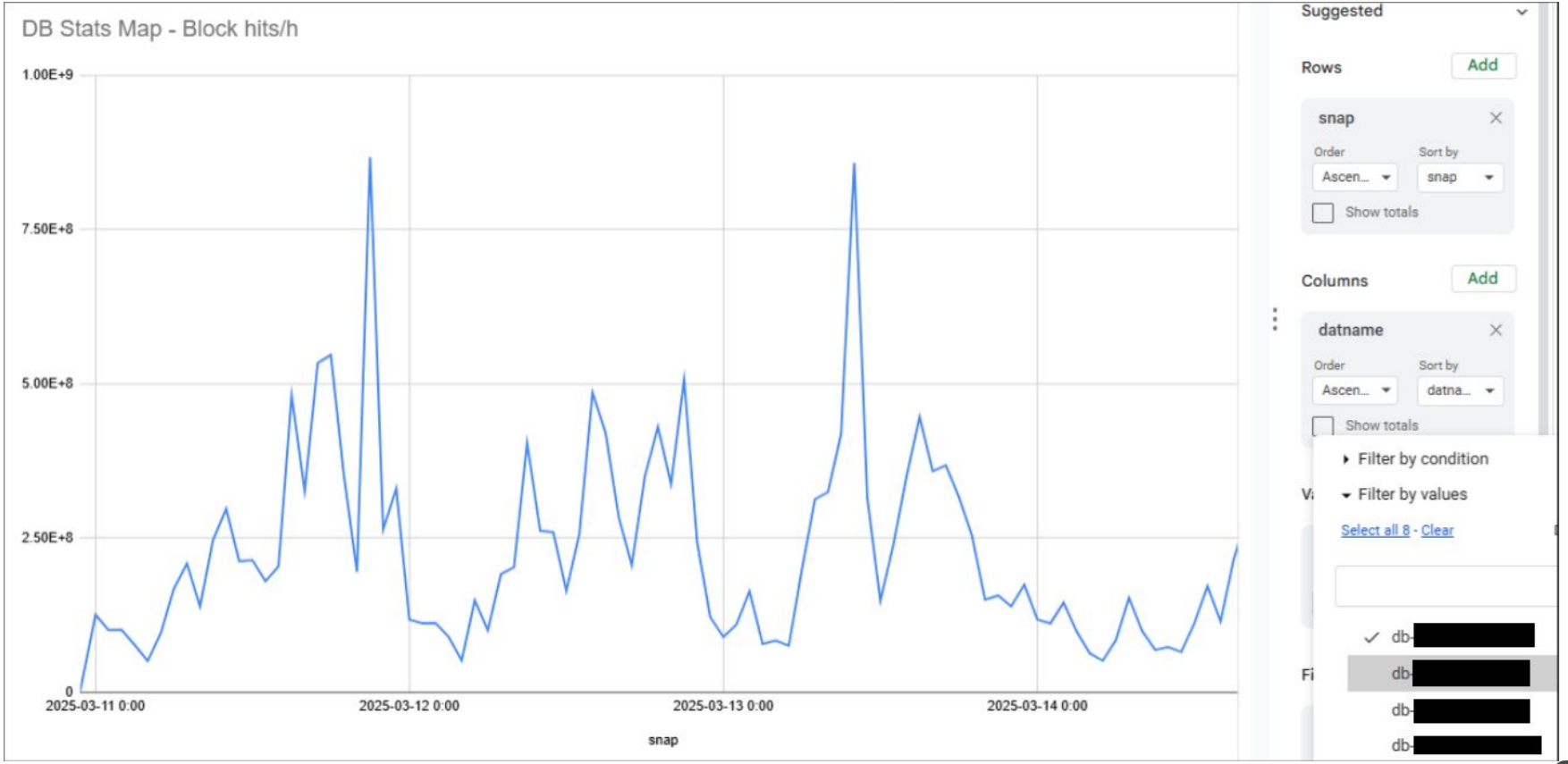
```
mydb=# \COPY connection_stats FROM '/tmp/all_conn_state.txt' DELIMITER ',' CSV HEADER;
COPY 816314

mydb=# \COPY db_stats FROM '/tmp/all_db_stats.txt
COPY 13990

mydb=# \i aas.sql
  host | snaps | cnt_sum | cnt_max | cnt_avg | ac
-------+-------+---------+---------+---------+---
 ins1  |   803 |     829 |       2 |   1.032 |
 inst2 |  2984 |    3953 |       5 |   1.325 |
 Inst3 |  5264 |   17539 |      12 |   3.332 |
 inst4 |  1198 |    1254 |       3 |   1.047 |
 inst5 |  5274 |   12056 |       6 |   2.286 |
 Inst6 |  3898 |    5088 |       5 |   1.305 |
...
```

```
# cat aas.sql

select host,
       count(1)                    snaps,
       sum(cnt)                    cnt_sum,
       max(cnt)                    cnt_max,
       round(avg(cnt),3)           cnt_avg,
       max(active_total)           active_max,
       round(avg(active_total),3) active_avg,
       max(max_in_db)              max_max_in_db
from (
  select host, snap, sum(count)  active_total,
         count(distinct datname) dbs,
         max(count) max_in_db,
         count(1) cnt
  from connection_stats
  where state='active' and datname!='postgres'
  group by host, snap
) group by host;
```

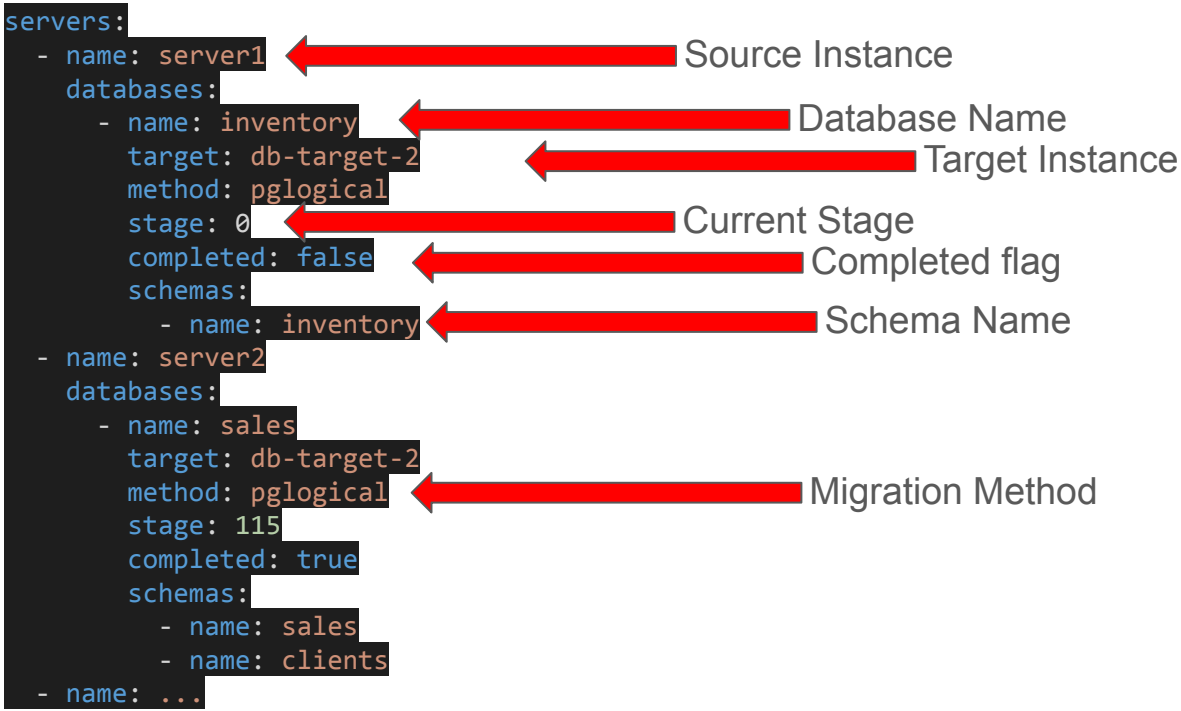
Capacity planning

```
mydb=# create view db_stats_snap_v as
select host, datname, snap, numbackends,
    xact_commit,    xact_commit    - lag(xact_commit   ) OVER (partition by host, datname ORDER BY snap) xact_commit_dt,
    xact_rollback,  xact_rollback  - lag(xact_rollback ) OVER (partition by host, datname ORDER BY snap) xact_rollback_dt,
    blks_read,      blks_read      - lag(blks_read     ) OVER (partition by host, datname ORDER BY snap) blks_read_dt,
    blks_hit,       blks_hit       - lag(blks_hit      ) OVER (partition by host, datname ORDER BY snap) blks_hit_dt,
    tup_returned,   tup_returned   - lag(tup_returned  ) OVER (partition by host, datname ORDER BY snap) tup_returned_dt,
    tup_fetched,    tup_fetched    - lag(tup_fetched   ) OVER (partition by host, datname ORDER BY snap) tup_fetched_dt,
    tup_inserted,   tup_inserted   - lag(tup_inserted  ) OVER (partition by host, datname ORDER BY snap) tup_inserted_dt,
    tup_updated,    tup_updated    - lag(tup_updated   ) OVER (partition by host, datname ORDER BY snap) tup_updated_dt,
    tup_deleted,    tup_deleted    - lag(tup_deleted   ) OVER (partition by host, datname ORDER BY snap) tup_deleted_dt,
    conflicts,      conflicts      - lag(conflicts     ) OVER (partition by host, datname ORDER BY snap) conflicts_dt,
    temp_files,     temp_files     - lag(temp_files    ) OVER (partition by host, datname ORDER BY snap) temp_files_dt,
    temp_bytes,     temp_bytes     - lag(temp_bytes    ) OVER (partition by host, datname ORDER BY snap) temp_bytes_dt,
    deadlocks,      deadlocks      - lag(deadlocks     ) OVER (partition by host, datname ORDER BY snap) deadlocks_dt
 from db_stats
 where datname not in ('template0','template1')
 order by host, datname, snap;

mydb=# \COPY (select * from db_stats_snap_v) TO '/tmp/db_stats_snap_v.csv' DELIMITER ',' CSV HEADER;
COPY 12742
```


Capacity planning – Sample graphs



Appendix – Migration automation – Config: migrate_db.yaml



Migration automation – Config: migrate_env.yaml

```yaml
general:
  postgresUser: "nelson"
  parallelStreams: 4
  superUser: dba
  ownerUser: db_owner
  replUser:  app_replica
  urlBase: prod.mydb.com
stages:
  - stage: 5
    description: "Create owner db_builder"
    script: "<binDir>/create_owner.sh -l <logFile> -h <target> -U <superUser> <ownerUser>"
    rollback: "<binDir>/drop_role.sh -h <target> -U <superUser> <ownerUser>"
    method: all
  - stage: 10
  ...
  - stage: 84
    description: "Load Database Data"
    script: "<binDir>/run_pgrestore.sh -l <logFile> -h <server> -U <superUser> -d <database> -a <target>"
    method: pgdump
  ...
  - stage: 90
    description: "Install pglogical on target"
    script: "<binDir>/install_pglogical.sh -l <logFile> -h <target> -U <superUser> -d <database> -r <replicaUser> -t"
    method: pglogical
```

Source DBA user

Parallel migrations

Run stage

Command

Replacement Variable

Migration Method

Migration automation - Reading / Writing config

- Use `yq`
  - `yq -r '.servers[] | select(.name == "db") | .databases[] | select(.name == "location") | .target'`
  - `yq -iy '(.servers[] | select(.name == "db") | .databases[] | select(.name == "location").stage) = 10'`
- Use `flock`
  - `flock db_config.yaml <yq command>`
- Parallel running
  - `RunMigrate <parameter1> <parameter2> &`
  - `wait -n`

Appendix – Migration automation – pglogical publisher creation

- Max of two publishers per database - one for insert-only schemas, other for the rest
- Code generated by the publisher creation step - easy to review generated SQL in case of issues

```
$ cat gensql/server1_db1_pub.sql
/*  ## Create publishers for pglogical replication on source host server1 for database db1 ##  */
/*     ### CRUD schemas - no log ###    */
do $$
declare
 ret oid;
begin
 if not exists (SELECT set_id FROM pglogical.replication_set WHERE set_name = 'db1_pub') then
    ret := pglogical.create_replication_set('db1_pub');
    RAISE NOTICE 'Replication set porch_pub created with OID %', ret;
 else
    RAISE NOTICE 'Replication set porch_pub already created - skipping creation';
 end if;
end $$;
/* schema1 schema */
GRANT USAGE ON SCHEMA schema1 TO app_replica;
GRANT SELECT,INSERT,UPDATE,DELETE ON ALL TABLES IN SCHEMA schema1 TO app_replica;
GRANT USAGE,SELECT ON ALL SEQUENCES IN SCHEMA schema1 TO app_replica;
SELECT pglogical.replication_set_add_all_tables('db1_pub', ARRAY['schema1']);
select pglogical.replication_set_add_all_sequences('db1_pub', ARRAY['schema1']);
/* schema2 schema */
...
```

Migration automation – pglogical publisher creation

- Max of two publishers per database - one for insert-only schemas, other for the rest
- Code ger

```
$ cat gensql/serv
/*  ## Create pub
/*     ### CRUD s
do $$
declare
 ret oid;
begin
 if not exists (S
    ret := pglogi
    RAISE NOTICE
 else
    RAISE NOTICE
 end if;
end $$;
/* schema1 schema
GRANT USAGE ON SC
GRANT SELECT,INSE
GRANT USAGE,SELEC
SELECT pglogical.
select pglogical.
/* schema2 schema
...
```

```
...
/*   ###  Insert only schemas ###   */
do $$
declare
 ret oid;
begin
 if not exists (SELECT set_id FROM pglogical.replication_set WHERE set_name = 'db1_log_pub') then
    ret := pglogical.create_replication_set(set_name := 'db1_log_pub',
                                            replicate_update:= false, replicate_delete:= false);
    RAISE NOTICE 'Replication set porch_log_pub created with OID %', ret;
 else
    RAISE NOTICE 'Replication set porch_log_pub already created - skipping creation';
 end if;
end $$;
/* schema1_log schema */
GRANT USAGE ON SCHEMA schema1_log TO app_replica;
GRANT SELECT,INSERT,UPDATE,DELETE ON ALL TABLES IN SCHEMA schema1_log TO app_replica;
GRANT USAGE,SELECT ON ALL SEQUENCES IN SCHEMA schema1_log TO app_replica;
SELECT pglogical.replication_set_add_all_tables('db1_log_pub', ARRAY['schema1_log']);
select pglogical.replication_set_add_all_sequences('db1_log_pub', ARRAY['schema1_log']);
/* schema2_log schema */
...
```

Pyt

Appendix - Migration automation - Schema and Data validation

- Initial version using Google Data Validation Tool (DVT)
  - https://github.com/GoogleCloudPlatform/professional-services-data-validator
  - Python code
  - Several executions per schema - a few seconds overhead, big total for large clusters
  - Errors casting column data types - fixed a few, but new DBs raised new errors
  - Requires custom config per source and target DB

- Final version using DVT (schema validation) and custom python script (data validation)
  - Schema validations implemented through external SQLs and DVT
    - Configurable and easy to deal with catalog version changes (i.e.: find all user created functions)
  - Data validations using hash over all table columns
    - Similar column casting errors as in DVT. Implemented all fixes in a single place
    - Special treatment for large tables
      - Limit to a few thousands sample rows (random PK)
      - Using TABLESAMPLE based on estimated table size

Migration automation – Schema validation

```
(venv) ncalero@test:/migra/bin$ python3 schema_compare.py source-inst db-name target-inst
Running validations for database: db-name on host source-inst
 Object Count Summary ...
 Schemas to validate: 'sales','customers','public'

 INFO - /mnt/dump/dvt/venv/bin/data-validation validate custom-query row -sc psql_test_db-name_source -tc
psql_test_db-name_target -sqf="/validation/modified_sql/source-inst/objectsummary_source_db-name.sql"
-tqf="/validation/modified_sql/source-inst/objectsummary_target_db-name.sql" --primary-keys="col1" --concat='*' -fmt 'csv'

 ... Warning: Some rows failed to validate
Validation Results:
source                                target                                validation_status
----------------------------------  ----------------------------------  -----------------
Schema count                    →   5  Schema count                    →   5  success
Extension count                 →   6  Extension count                 →   6  success
View count                      →   1  View count                      →   1  success
Table count                     →   6  Table count                     →   7  fail
Materialized View count         →   0  Materialized View count         →   0  success
Index count                     →  29  Index count                     →  29  success
Partition count                 →   7  Partition count                 →   6  fail
Trigger count                   →  18  Trigger count                   →  18  success
Sequence count                  →   8  Sequence count                  →   8  success
Primary Key count               →  13  Primary Key count               →  13  success
Foreign Key count               →   5  Foreign Key count               →   5  success
Unique Constraint count         →   7  Unique Constraint count         →   7  success
Check Constraint count          →   1  Check Constraint count          →   1  success
Userdefined Function count      →   6  Userdefined Function count      →   6  success
```

Migration automation – Data validation

Code to cast column data types:

```
def _process_value(self, col, value):

   if isinstance(value, (datetime, time.struct_time, dt_time)):
        return value.isoformat()

   elif isinstance(value, (DateTimeTZRange, Decimal, NumericRange, date, timedelta, list, DateRange)):
        return str(value)

   elif isinstance(value, memoryview):
        return base64.b64encode(value.tobytes()).decode('utf-8')

   else:
        return value
```


Migration automation – Data validation

Code to get random PK for large tables:

```
    def _get_random_primary_keys(self, conn, schema, table, pk_columns, sample_perc):
        cursor = conn.cursor()
        pk_column_str = ", ".join(pk_columns)
        where_clause = " AND ".join([f"{col} IS NOT NULL" for col in pk_columns])
        sql = f"""
                SELECT {pk_column_str}
                FROM {schema}.{table} TABLESAMPLE SYSTEM(%s)
                WHERE {where_clause}
                LIMIT %s;
        """
        fetch_limit = max(2 * self.sample_size, 1000)
        self._execute_sql(cursor, sql, conn, (sample_perc,fetch_limit,))
        if len(pk_columns) == 1:
            all_pks = [row[0] for row in cursor.fetchall()]
        else:
            all_pks = [tuple(row) for row in cursor.fetchall()]

        cursor.close()
        return random.sample(all_pks, min(len(all_pks), self.sample_size))
```


Migration automation – Data validation

Code to get random PK for large tables:

```
    def _get_random_primary_keys(self, conn, schema, table, pk_columns, sample_perc):
        cursor = conn.cursor()
...
   estd_rows = self._get_table_rows_est(source_conn, schema, table)
   safety_factor = Decimal(3)
   v_sample_perc = (Decimal(self.sample_size) / max(estd_rows,1)) * 100 * safety_factor
   v_sample_perc = max(Decimal('0.00001'), min(Decimal('100.0'), v_sample_perc))

   logging.info(f"....  getting random PKs in source (rows={estd_rows}  sample={v_sample_perc:.3f}%)")
   source_pks = self._get_random_primary_keys(source_conn, schema, table, pk_columns, v_sample_perc)
   if not source_pks:
      logging.info(f"No data found in {schema}.{table} or sample size is zero.")
      status = "OK (No Data)"
      return status, time.time() - start_time
...
            all_pks = [tuple(row) for row in cursor.fetchall()]

        cursor.close()
        return random.sample(all_pks, min(len(all_pks), self.sample_size))
```