intrbiz

To PostgreSQL

And Beyond...

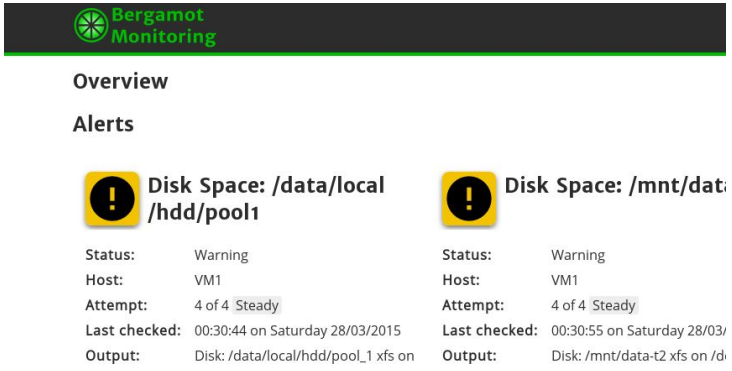
NOVALUG Meetup 2021

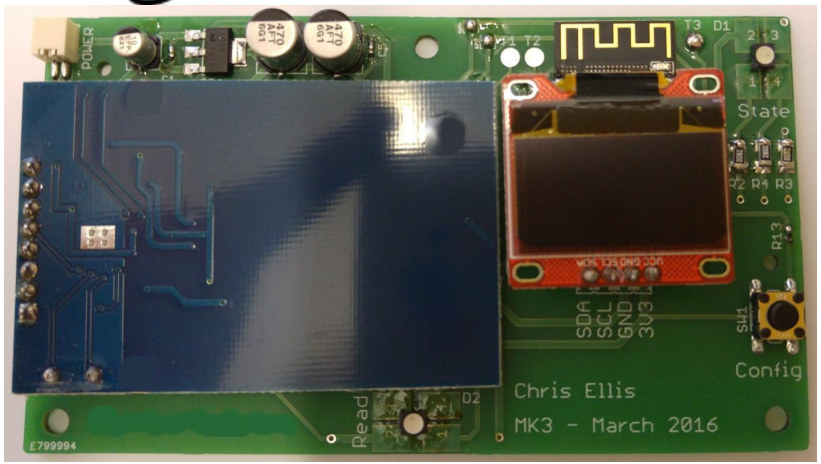
Chris Ellis - @intrbiz

intrbiz

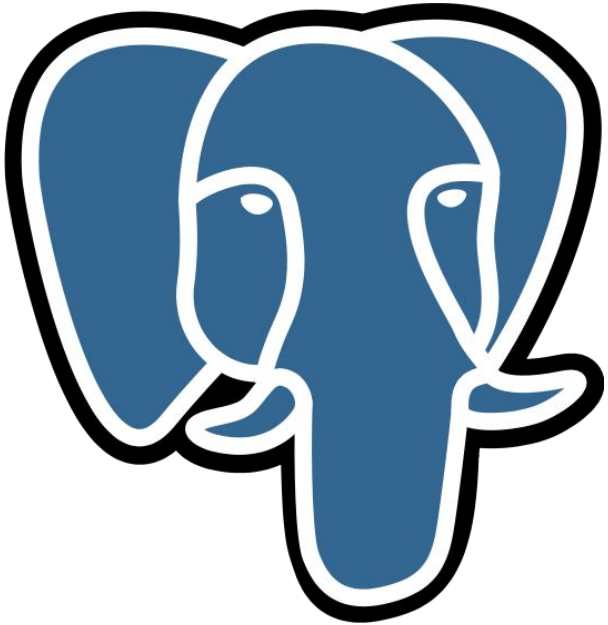
Hello!

Bergamot Monitoring





openSUSE



About PostgreSQL

- PostgreSQL is an advanced open source object-relational database
  - Code goes back a long time, Postgres started in 1982, PostgreSQL came along in 1996
- Well regarded for
  - Reliability and data integrity
  - Feature robustness and correctness
  - Performance and scalability
  - Features available and SQL spec adherence
  - Extensibility, flexibility and a long history of innovation
  - Supported on a wide range of platforms
- Won DB-Engines Database of the year 3 times, including 2020
- More: https://www.postgresql.org/

Getting Started: Installing

- You'll find PostgreSQL in most distro packages
- PGDG provide 'official' packages for a range of distros
  - https://www.postgresql.org/download/
- Installing is a case of:
  - `zypper in postgresql13 postgresql13-server postgresql13-contrib postgresql13-llvmjit`
  - `systemctl start postgres`

Getting Started: Connecting

- Easiest way to connect first time round is via psql, the PostgreSQL SQL CLI
  - `sudo -u postgres psql`
    - This will connect to PostgreSQL as the `postgres` superuser
  - pgAdmin4 is the defacto graphical client
  - OmniDB, DBeaver, Navicat, are alternatives
- A PostgreSQL server hosts many databases (so called database cluster)
  - Each database is isolated from each other
    - You cannot query across databases
    - Databases still have schemas inside for namespacing
  - A default database `postgres` exists and a couple of template databases
    - Do not use these, create a database for your application first

cellis : bash — Konsole <2>

File Edit View Bookmarks Settings Help

```
chris-desktop:/home/cellis # sudo -u postgres psql
psql (13.1, server 13.1)
Type "help" for help.

postgres=# CREATE ROLE demo LOGIN NOSUPERUSER;
CREATE ROLE
postgres=# \password demo
Enter new password:
Enter it again:
postgres=# CREATE DATABASE demo OWNER demo;
CREATE DATABASE
postgres=# \q
```

cellis : bash

Getting Started: Basic Config

- PostgreSQL has 2 main configuration files
  - These are all per 'database cluster'
  - These files are typically in the data directory, some distros move them to /etc
- `postgresql.conf`
  - This is the main PostgreSQL configuration file
  - Changing some options will require a restart of the PostgreSQL server processes
- `pg_hba.conf`
  - This is the host based access configuration file
    - It controls which network clients can connect and how they should authenticate
  - Changes to this file don't require a restart of the PostgreSQL server processes
    - It's reread if the postmaster process receives a SIGHUP

Getting Started: Config Gotchas

- Out of the box config is conservative and safe
  - Enable remote access: `listen_addresses`
  - Maximum number of clients: `max_connections`
  - Increase logging
- Constraints on replication / backups by default
  - Worth raising `max_wal_senders`
  - Worth raising the `wal_level`
- Remote access
  - Probably want to update `pg_hba.conf` to allow remote users to connect

Getting Started: Perf Gotchas

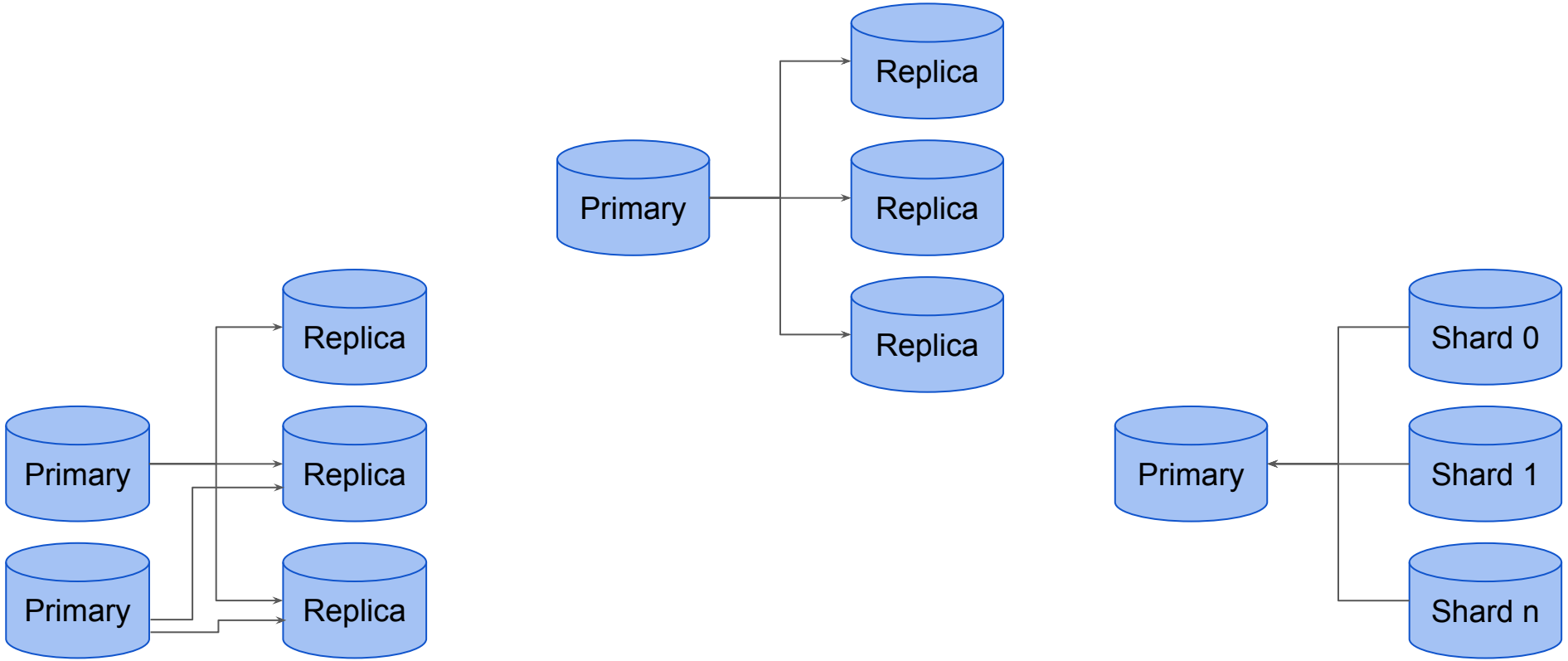
- A few basic settings you look at:
  - `shared_buffers`
    - Rule of thumb is 25% of server RAM for a dedicated DB server
  - `min_wal_size` / `max_wal_size`
    - You want to aim for consistent I/O under load
  - `work_mem`
    - Don't set this too high, it's allocated multiple times per connection
  - `random_page_cost`
    - For SSDs lower this to around 1.1
  - `fsync`
    - You might find bad advice about turning this off, do not! Ever!
- Checkout: https://pgtune.leopard.in.ua/

Getting Started: Perf Gotchas

- Use SSDs!
- Filesystems
  - Most modern are good: XFS, EXT4, BTRFS all offer good performance
  - Don't use NFS
- Kernel
  - Disable memory overcommit on dedicated servers and reduce swapiness (or no swap)
  - Tune the page cache dirty writing - want consistent I/O rather than spiky
- Benchmark:
  - `pgbench`
  - `bonnie++ / sysbench / fio`

Getting Started: Upgrades

- Updating between major versions of PostgreSQL is not so straightforward
  - Major versions are: 9.5, 9.6, 10, 11, 12, 13
  - Minor changes are simple: 13.1 to 13.2, 9.5.1 to 9.5.2 is just install and restart
- The PostgreSQL data directory (where your database resides on disk) is not compatible between major releases
  - You cannot start PostgreSQL 13 with a 12 data directory
- However there is `pg_upgrade` which solves most issues
  - This will convert the data directory from major version to major version
  - Can actually be very fast to run, even on large databases
  - It is an offline tool!


Primary
Replica
Replica
Replica
Primary
Primary
Replica
Replica
Replica
Primary
Shard 0
Shard 1
Shard n

Replication: Streaming Replication

- PostgreSQL has asynchronous and synchronous replications built in
- This lets you replicate all databases and changes between servers
  - All servers must be of the same major version
- It works by streaming the Write Ahead Log (WAL) from the primary server to all secondary servers and replaying the changes
- Secondary server can run in hot standby mode, this allows them to execute read only queries
- By default is asynchronous, so there can be a lag between writing to the primary and data being available on a replica

Replication: Synchronous Streaming Replication

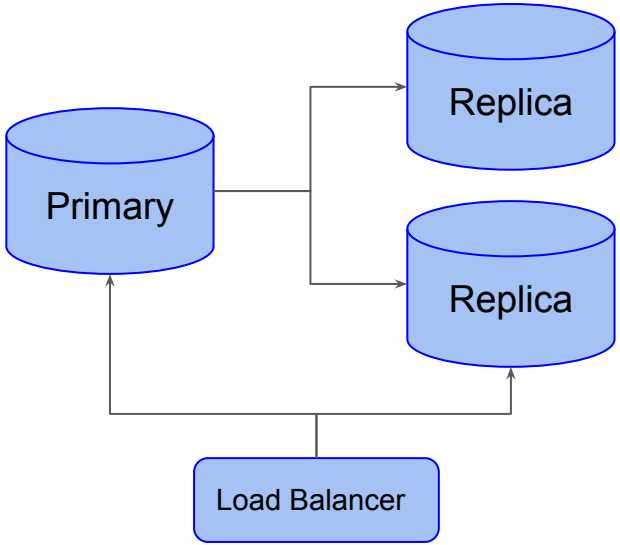
- Will block client transaction commit until all replicas have replayed it.
  - This will have some performance impact
- Also supports quorum commit
  - Meaning a transaction is committed when a majority or replicas have replayed it
- Can also be enabled on a transaction by transaction basis

Replication: Logical Replication

- A scalpel compared to the sledgehammer of streaming replication
  - Lets you replicate at the table by table level
  - Performs change data capture by decoding the WAL stream
  - Handles the initial data synchronization
- Can replicate between different major versions of PostgreSQL
  - Can be used to perform live database upgrades
- Lets you replicate to other data systems
  - You can stream logical changes anyway you like
    - For example into Kafka or Pulsar
- Fair bit newer that streaming replication
  - Since PostgreSQL 10

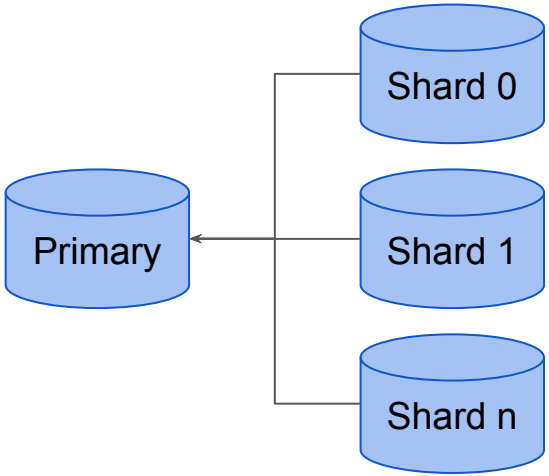
Replication: Clustering / High Availability

- PostgreSQL doesn't have any clustering capabilities out of the box
  - Again clustering databases is a huge and complex topic
- However replication gives you the building blocks
  - External tools need to provide:
    - Failover between nodes
    - Load balancing between nodes
  - Tools built around this like:
    - Patroni
    - pg_autofailover



Replication: Clustering / Sharding

- No out of the box sharding
  - It's a pretty niche use case, you probably don't need it
  - Scale up first
  - Distributed ACID is extremely difficult (CAP theorem)
- But there are approaches
  - pl/proxy
  - Foreign Data Wrapper (FDW)
  - Logical Replication
    - BDR
  - Citus
  - PostgreSQL XL
  - Application Layer




SQL

JSON / JSONB

- The growth of NOSQL was mainly around document databases, which often espoused the ability to store data without a schema
- PostgreSQL reacted by adding a JSON data type and various support functions
  - Originally this was just stored as text (but validated as JSON)
  - Functions to convert rows to JSON object, construct JSON nodes via SQL
- Then came along JSONB
  - Stores schemaless JSON data in a binary format
  - Fully index accelerated
  - Basically MongoDB in a column

JSON / JSONB

```
CREATE TABLE bergamot.check_metadata (
   check_id UUID NOT NULL PRIMARY KEY,
   meta     JSONB
);

CREATE INDEX metadata_idx ON
bergamot.check_metadata USING GIN(meta);
```

JSON / JSONB

```
-- Lets pull some fields out (as text)
SELECT *, meta ->> 'command'
FROM bergamot.check_metadata;

-- As JSON
SELECT *, meta -> 'parameters'
FROM bergamot.check_metadata;
```

JSON / JSONB

```
-- Lets search for something in the JSON
SELECT *
FROM bergamot.check_metadata
WHERE meta ->> 'name' = 'test1';

-- Looks like that should work, and it
does, just very slowly!  Since this query
can't use the index
```

JSON / JSONB

```
-- Lets search for something in the JSON
SELECT *, meta ->> 'command'
FROM bergamot.check_metadata
WHERE meta @> '{"name": "test_a"}'::JSONB;

-- This uses the contains operator and is
fully index accelerated! 3ms vs seconds
```

JSON / JSONB

```
-- Lets search for something in the JSON
SELECT *, meta ->> 'command' AS command
FROM bergamot.check_metadata
WHERE meta @@ '$.parameters[0] == "A"';

-- This uses a JSONPath expression to
search deep into the JSON structure
```

JSON / JSONB

```
-- A whole table as a single JSON tree
SELECT json_agg(row_to_json(h.*))
FROM bergamot.host h;

-- Build an object
SELECT json_build_object('name', 'test',
'command', 'check_thingy', 'parameters',
ARRAY['A', 'B', 'C']);
```

JSON / JSONB

```
-- Or nest relation in an object structure
SELECT json_build_object(
    'id', id,
    'name', name,
    'summary', summary,
    'services', (SELECT json_agg(row_to_json(s.*)) FROM
bergamot.service s  WHERE s.group_ids @> ARRAY[g.id])
)
FROM bergamot.group g;
```

Full Text Search

- Provides text based searching inside PostgreSQL
  - Lets your search for words within text documents
  - Provides ranking capabilities for sorting which document best matches your search
  - Search for multiple words, phrase search
  - Provides stemming and snowballing
  - Supports multiple languages

- Originally started off as an extension, eons ago
  - Introduced pluggable index support to PostgreSQL
    - PostgreSQL does a lot more than just BTrees

Full Text Search

```
CREATE TABLE bergamot.check_search (
    check_id    UUID,
    check_type  TEXT,
    vector      TSVECTOR
);

CREATE INDEX check_search_idx ON
bergamot.check_search USING GIN (vector);
```

Full Text Search

```
-- Lets populate some text search vectors
INSERT INTO bergamot.check_search
SELECT
id,
'service',
setweight(to_tsvector('english', coalesce(summary,'')), 'A')   ||
setweight(to_tsvector('english', coalesce(description,'')), 'C')

FROM bergamot.service;
```

Full Text Search

```
SELECT *
FROM bergamot.service c
JOIN bergamot.check_search s ON
   (c.id = s.check_id)
WHERE
s.vector @@ to_tsquery('Postgresql & version')
;
```

Going global: PostGIS

- PostGIS is a complete geographic information system for PostgreSQL
  - Add geographic and geometry data types
  - Add a huge number of functions for working with these data types
  - Provides external tools for loading data into PostgreSQL from various geographic data formats

- PostGIS is probably the biggest and most widely used external extension
  - This means you will need to install the libraries separately

cellis : bash — Konsole <2>

File Edit View Bookmarks Settings Help

```
chris-desktop:/home/cellis # zypper in postgresql12-postgis postgresql12-postgis-utils
...
chris-desktop:/home/cellis # sudo -u postgres psql
psql (13.1, server 13.1)
Type "help" for help.

postgres=# \c demo
postgres=# CREATE EXTENSION postgis;
postgres=# \q
```

cellis : bash

Going global: PostGIS - Search within a distance

```
SELECT date_trunc('month', r.day) AS month,
  avg(r.kwh), min(r.kwh), max(r.kwh)
FROM reading r
JOIN meter m ON (m.id = r.meter_id)
JOIN postcode p ON st_dwithin(m.location,
                              p.location, 2000)
WHERE p.postcode = 'SY2 6ND'
GROUP BY 1;
```

intrbiz

SQL

In 2021

chris@intrbiz.com

Modern SQL: WITH queries (CTEs)

```
WITH checks AS (
    SELECT id, name FROM bergamot.host
    UNION
    SELECT id, name FROM bergamot.service
)
SELECT c.*, s.ok, s.status, s.output
FROM checks c -- Use the CTE
JOIN bergamot.check_state s ON (c.id = s.check_id);
```

Modern SQL: Recursive CTEs

```
WITH RECURSIVE group_graph(id) AS (
    SELECT g.id, g.name, ARRAY[g.name] AS path
    FROM bergamot.group g
    WHERE g.id = '59bec13b-8a3f-42d3-b6d4-9387ec160a5e'
UNION
    SELECT g.id, g.name, gg.path || ARRAY[g.name] AS path
    FROM bergamot.group g, group_graph gg
    WHERE g.group_ids @> ARRAY[gg.id]
)
SELECT * FROM group_graph;
```

Modern SQL: FILTER

```
-- Filter makes it really easy to compute aggregates
   with different criteria

SELECT
  count(*) FILTER(WHERE s.status = 2) AS ok_count,
  count(*) FILTER(WHERE s.status = 3) AS warning_count,
  count(*) FILTER(WHERE s.status = 4) AS critical_count
FROM bergamot.check_state s;
```

Modern SQL: Window Functions

Modern SQL: Window Functions

```
SELECT
 day,
 energy,
 energy - coalesce(lag(energy)
    OVER (ORDER BY day), 0) AS consumed
FROM iot.meter_reading
ORDER BY day;
```

Modern SQL: Window Functions

```
WITH consumption AS (
  … from previous slide …
)
SELECT *, sum(consumed) OVER
 (PARTITION BY date_trunc('week', day))
   AS weekly_total
FROM consumption;
```

Modern SQL: Window Functions

```
SELECT *, avg(consumed) OVER
 (ORDER BY day
  ROWS BETWEEN 2 PRECEDING
  AND CURRENT ROW)
   AS weekly_total
FROM consumption;
```

Modern SQL: Window Functions - Custom Aggregates

```
CREATE FUNCTION last_agg(anyelement, anyelement)
RETURNS anyelement LANGUAGE SQL IMMUTABLE STRICT AS $$
        SELECT $2;
$$;

CREATE AGGREGATE last (
        sfunc = last_agg,
        basetype = anyelement,
        stype = anyelement
);
```

Modern SQL: Window Functions - Custom Aggregates

```
CREATE FUNCTION last_agg(anyelement, anyelement)
RETURNS anyelement LANGUAGE SQL IMMUTABLE STRICT AS $$
        SELECT $2;
$$;

CREATE AGGREGATE last (
        sfunc = last_agg,
        basetype = anyelement,
        stype = anyelement
);
```

So Long And Thanks For All The Fish

- Thanks for listening
  - I hope I didn't bore you too much

- Questions?