

# PostgreSQL for IoT

## The Internet Of Strange Things

PGCONF.EU 2019 - Milan

Chris Ellis - @intrbiz

# Hello!

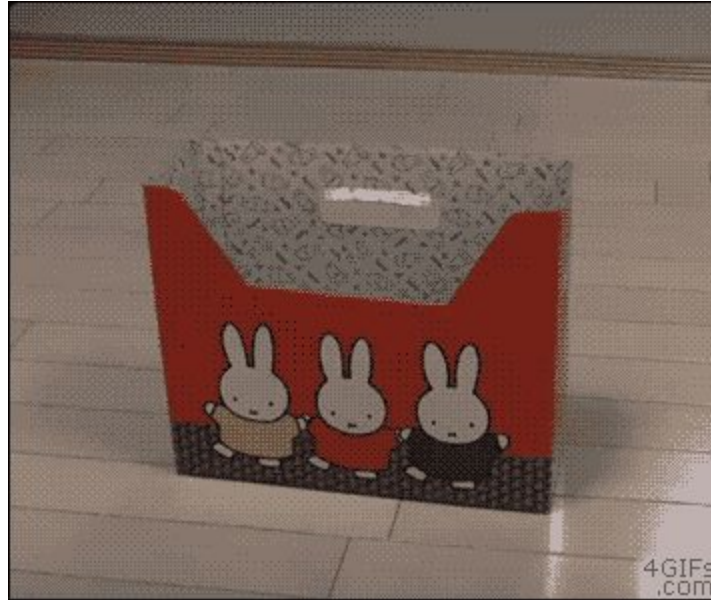
- I'm Chris
  - IT jack of all trades, studied Electronic Engineering
- Been using PostgreSQL for about 15 years
- Very much into Open Source
  - Started Bergamot Monitoring - open distributed monitoring
- Worked on various PostgreSQL systems
  - Connected TV Set top boxes
  - Smart energy meter analytics
  - IoT Kanban Board
  - IoT CHP Engines
  - Mixes of OLTP and OLAP workloads
  - Scaled PostgreSQL in various ways for various situations



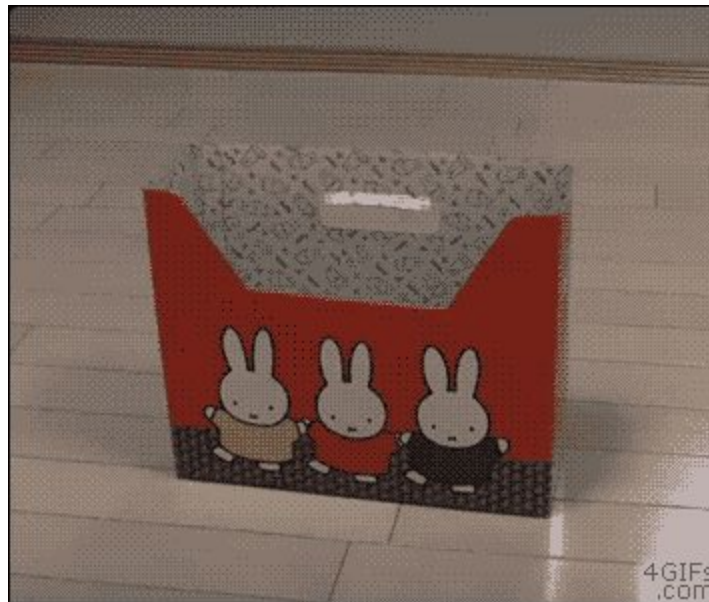
IoT



# One size fits all?

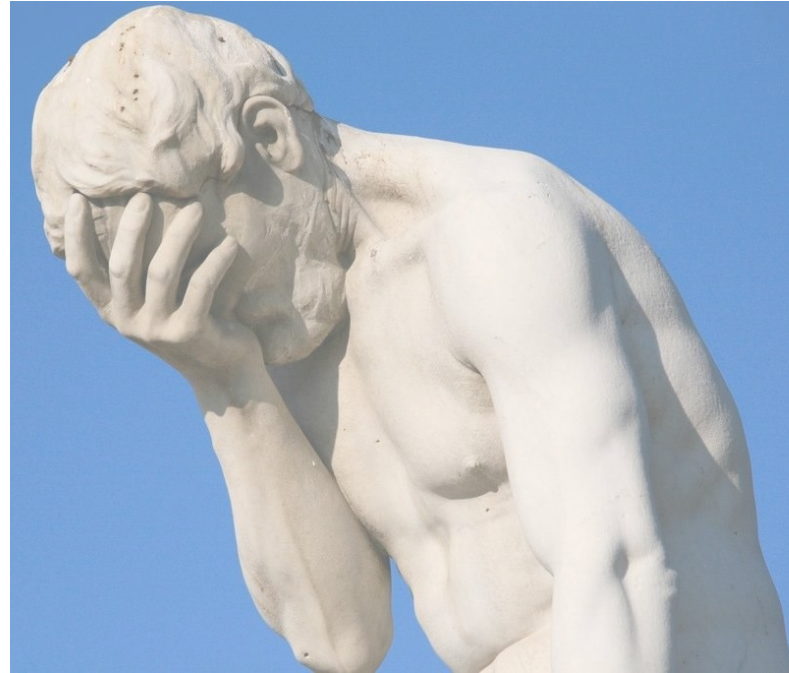


# One size fits all?



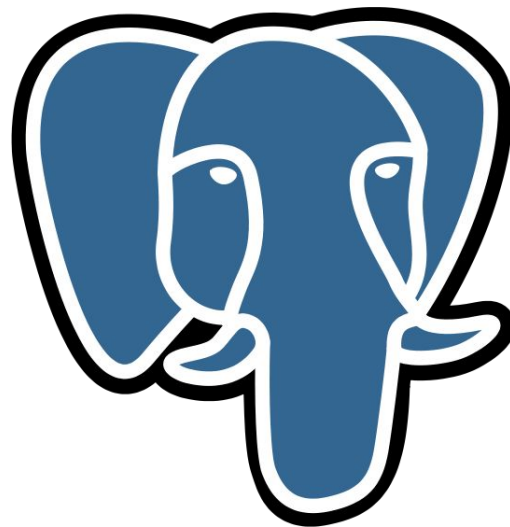
# Time series databases

- Lots of specialised time series datastores
  - Single use case solutions
  - Have their own querying languages
  - Limited data types



# Why PostgreSQL?

- The same reason I constantly go back to PostgreSQL
  - We don't call it the `world's most advanced Open Source relational database` without just cause
  - It's flexible
  - It's extensible
  - It puts up with you
  - It cares
- IoT is not a simple, one size fits all problem
  - It's not just time series data
  - I find single solution data stores, a bit, pointless



# Why PostgreSQL?

- PostgreSQL makes it easy to combine your time series data with other data
  - You know: a join!
- Find me the average energy consumption of Shropshire?
- Find me the average energy consumption for 4 bed houses during the summer?
- Find me the average, min, max energy consumption for 4 bed houses during summer in Shropshire for a half hourly period?
- What is the average energy consumption for houses within x miles of my house?

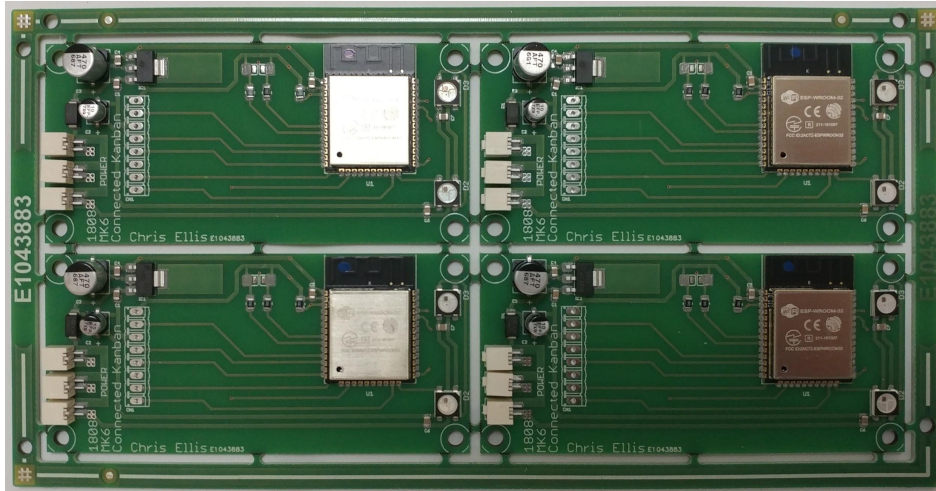


"Where you must go; where the path of the One ends."



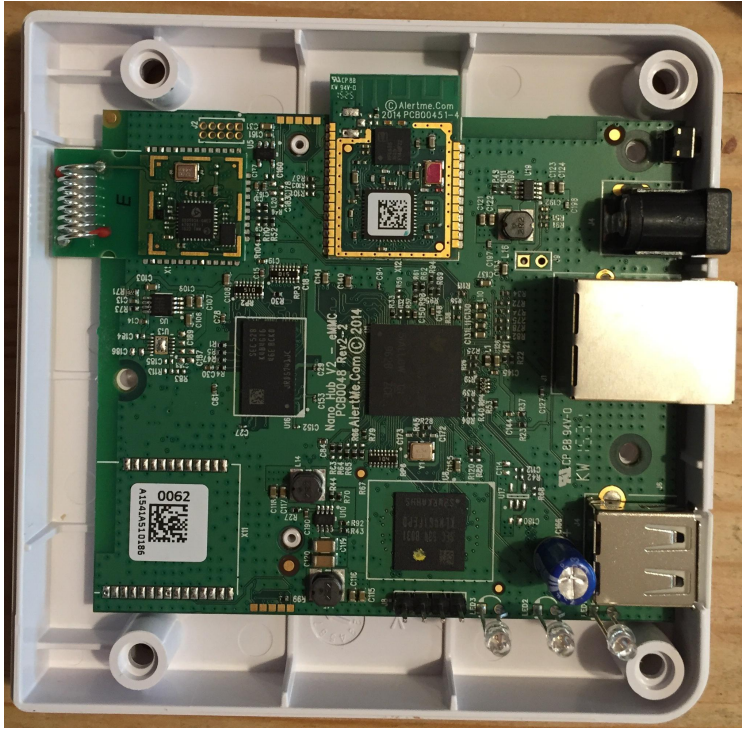
# "Where you must go; where the path of the One ends."

- The source of your data is usually a small embedded system
  - Can have very variable capabilities
    - From not enough to far to much



- ESP-32
  - Dual core 32bit @ upto 240MHz
  - 520KiB SRAM (D&I)
  - Typically 4MiB SPI Flash ROM
  - WiFi, TCP/IP stack
  - Runs FreeRTOS

"Where you must go; where the path of the One ends."



- Some devices can be pretty powerful with good RAM and storage
- Smart Home Hub
  - Single Core 1GHz ARM Cortex-A8
  - 512 MiB RAM
  - 4 GiB Flash eMMC Storage
  - WiFi + Ethernet
  - Zigbee
  - Runs Linux

"Where you must go; where the path of the One ends."

- Other devices can be even stranger
  - Whole string of controllers and modules
  - Fairly busy control system, connectivity is not a priority



- Industrial Control
  - Single Core 200MHz ARM7
  - 128 MiB RAM
  - >8GB SD Card
  - Ethernet
  - Lots of CAN
  - Runs a RTOS, hard real time
  - Doing other very important things

# Collecting Data



# Collecting Data - Device $\longleftrightarrow$ Platform

- Probably using MQTT between device and platform
  - Seen AMQP to platform (terrible idea)
    - And some strange reinventions of TCP over UDP and DNS
  - Most likely sending binary data, especially if low end device
- Consumer devices might need to be careful of
  - Bandwidth utilisation
  - Power consumption
- Devices operating in remote environments
  - Need to be careful with battery usage
    - Eg: Gas meters must be battery powered
  - GPRS backhaul, slow, expensive during daytime

# Collecting Data - Device $\longleftrightarrow$ Platform

- Be selective about how you send data
  - A lot of use cases don't need low latency real time data feeds
    - Can switch to a fast mode when you need it
  - In the cloud you often get charged per message
    - Cheaper to send 1 big message than lots of small messages
- Business model
  - IoT products are quite often hero products, one off income (especially in consumer)
  - Yet you have recurring directly coupled costs
- Can be difficult to authenticate devices
  - TLS client auth often used, certs can be extracted and usually cover lots of devices
  - Low end devices harder to do certificates
  - Huge risk of people being able to fake data or do fun things

# Storing Data





## Storing Data

```
CREATE TABLE iot.alhex_reading (  
    device_id    UUID NOT NULL,  
    read_at     TIMESTAMP NOT NULL,  
    temperature REAL,  
    light       REAL,  
    PRIMARY KEY (device_id, read_at)  
);
```

## Storing Data - Range Types

```
CREATE TABLE iot.alhex_reading (  
    device_id    UUID NOT NULL,  
    read_range   TSRANGE NOT NULL,  
    temperature  REAL,  
    light        REAL,  
    PRIMARY KEY (device_id, read_range)  
);
```

## Storing Data - Metadata

```
CREATE TABLE iot.alhex_reading (  
  device_id    UUID NOT NULL,  
  read_range   TSRANGE NOT NULL,  
  temperature  REAL,  
  meta         JSONB,  
  PRIMARY KEY (device_id, read_at)  
);
```

## Storing Data - Rolling On Up

```
CREATE TABLE iot.daily_reading (  
    meter_id          UUID NOT NULL,  
    read_range        DATERANGE NOT NULL,  
    energy            BIGINT,  
    energy_profile    BIGINT[],  
    PRIMARY KEY (device_id, read_at)  
);
```

## Storing Data - Rolling On Up

t_xmin	t_xmax	t_cid	t_xvac	t_ctid	t_infomask 2	t_infomask	t_hoff
4	4	4	4	6	2	2	1

**24 bytes**

device_id	read_at	temperature	light
16	8	4	4

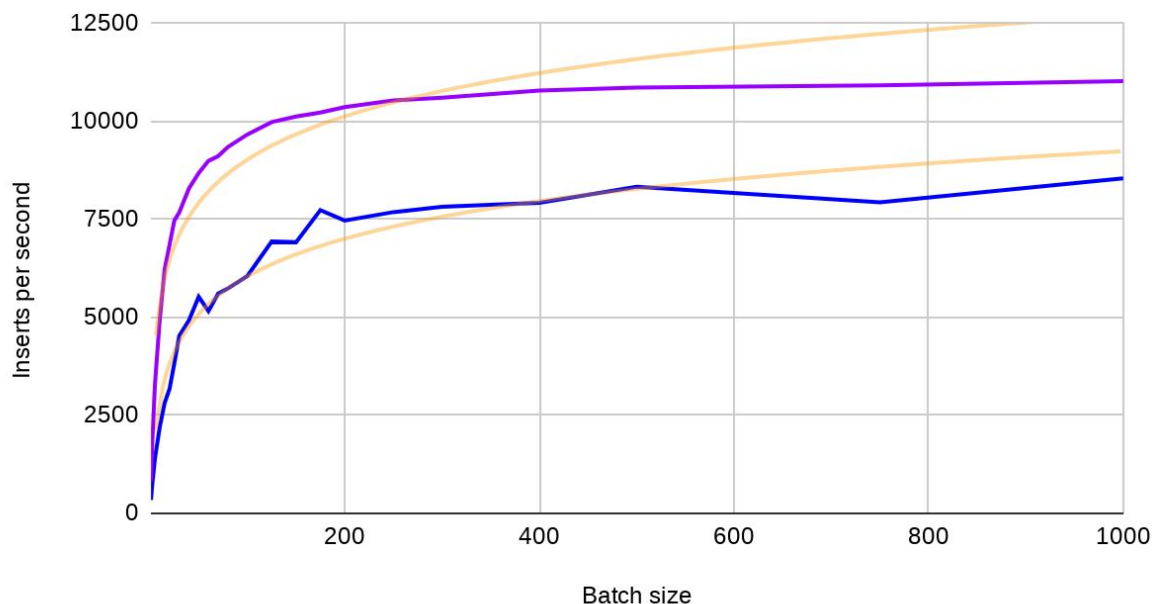
**32 bytes**

# Loading Data



# Loading Data - Batching

Loading Performance (Batching)



- Load in batches
- Don't use autocommit
- Batching ramps up fast:
  - Autocommit: 300 /s
  - Batch of 10: 2k2 /s
  - Batch of 50: 5k5 /s
  - Batch of 100: 6k /s
  - Batch of 300: 8k /s
- Batching gives ~ 20x performance gain

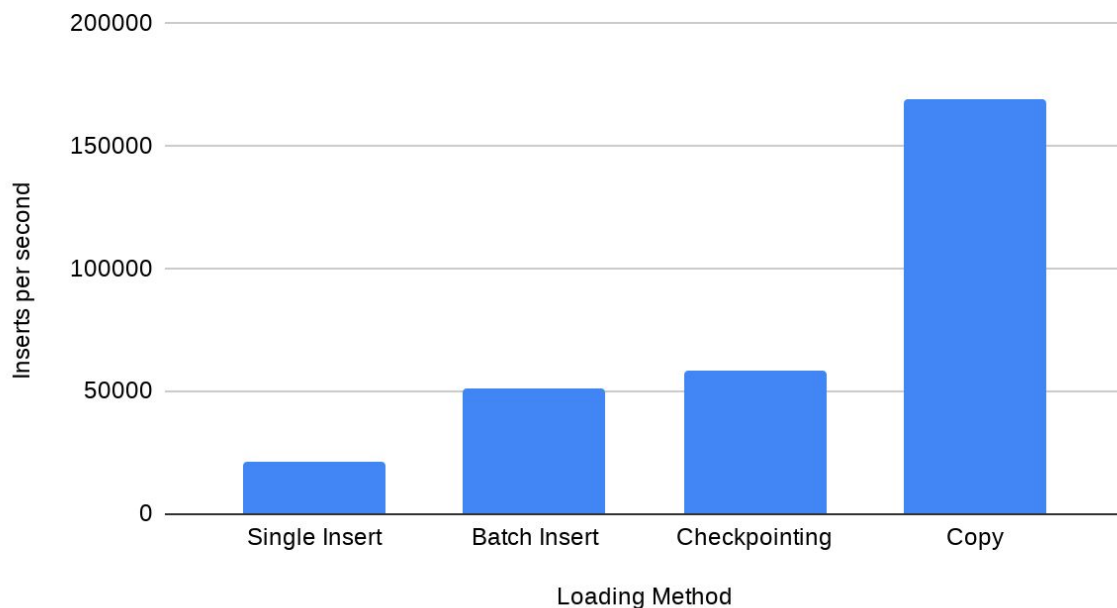
# Loading Data - Batching

```
connection.setAutoCommit(false);
try {
    try (PreparedStatement stmt = connection.prepareStatement("INSERT INTO ....")) {
        for (T record : batch) {
            stmt.setString(1, record.getId().toString());
            stmt.setTimestamp(2, record.getTimestamp());
            stmt.setFloat(3, record.getTemperature());
            stmt.addBatch();
        }
        stmt.executeBatch();
    }
    connection.commit();
} catch (SQLException e) {
    connection.rollback();
} finally {
    connection.setAutoCommit(true);
}
```



# Loading Data - Comparing Loading Methods

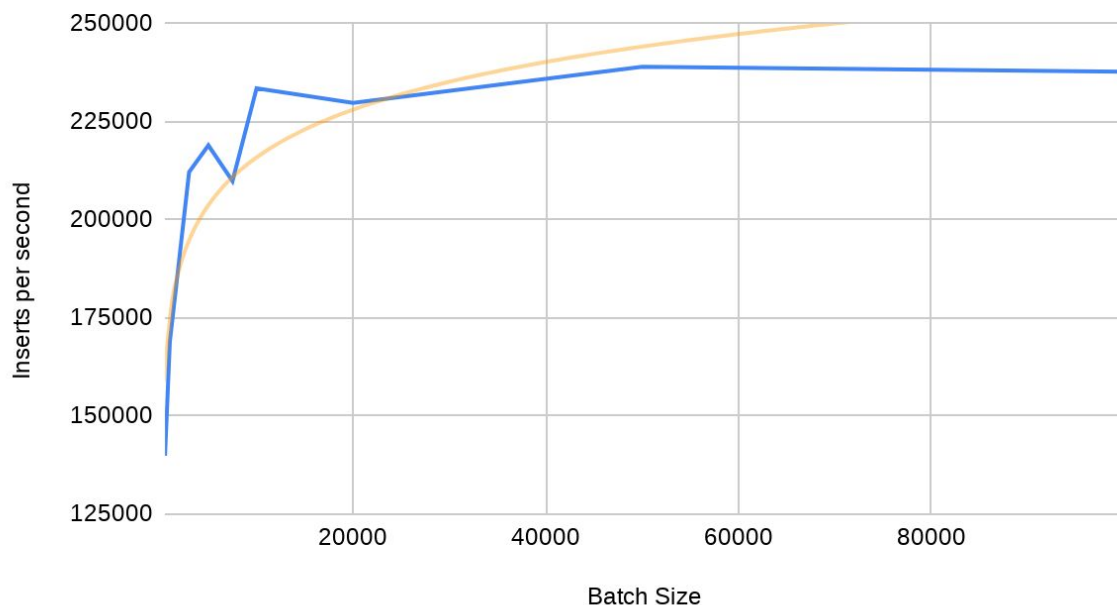
Loading Performance (Method)



- Batched inserts offer a big gain over single insert statements
- Copy has a huge speed up over even batched inserts with the same batch size
- Checkpointing is useful to keep latency consistent

# Loading Data - Copy Performance

Loading Performance (Copy)



- Copy starts fast and ramps up quickly with batch size
-

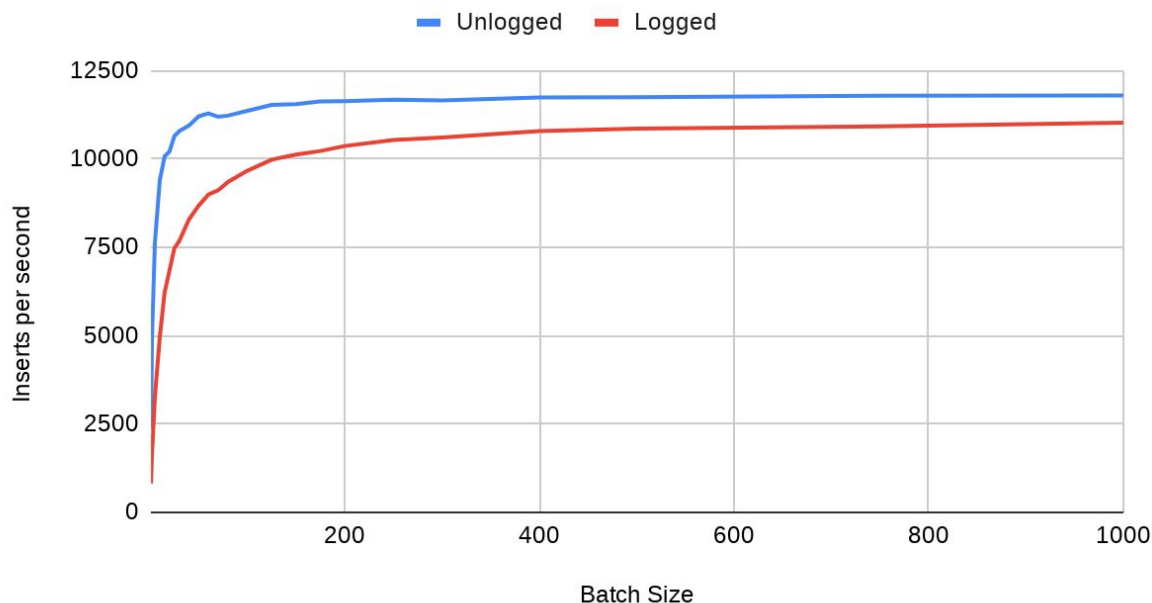
## Loading Data - ON CONFLICT



- Use ON CONFLICT
- Your data will be crap
  - Duplicate PKs
  - Out of order
- Nothing worse than having your batch abort
  - Need to deal with savepoints, application buffers
  - Gets rather complex

# Loading Data - Unlogged

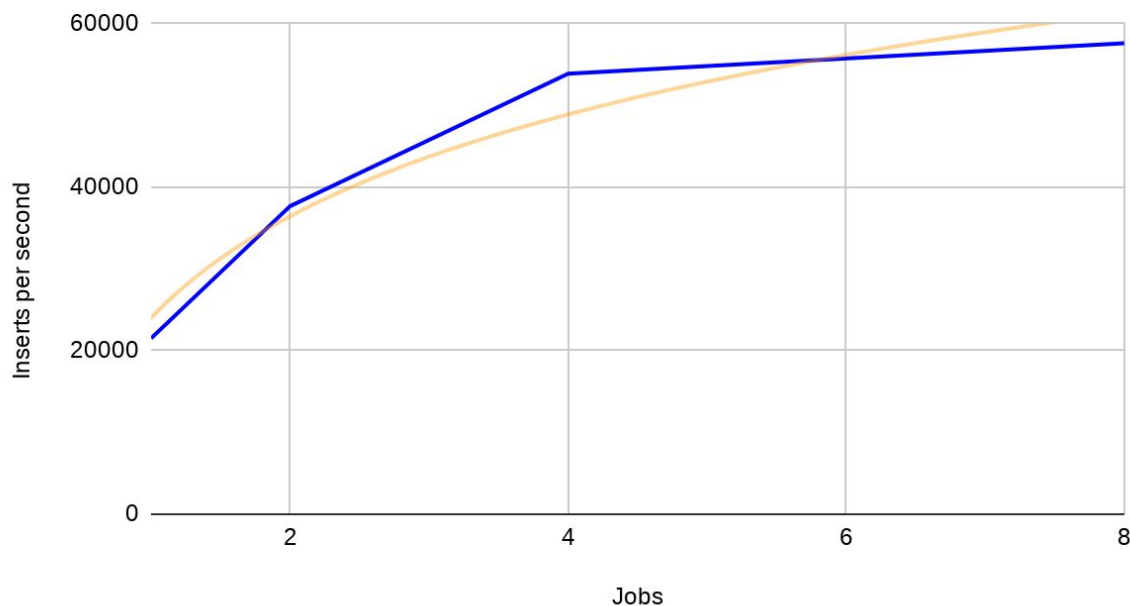
Loading Performance (Unlogged)



- UNLOGGED tables will ramp up faster than LOGGED tables with respect to batch sizes
- Little improvement over optimized batch loading

# Loading Data - Parallel

Loading Performance (Jobs)



- Loading in parallel will let you push more in
- Roughly linear until you hit CPU or Storage limits

# Loading Data - Never Sleeping

- IoT data is often constant, never sleeping, never lets up
  - Also insert / append only doesn't trigger AutoVac, your tables don't get ANALYSEd
- This can really stresses replication
  - Regardless of sync vs async replication
  - You need to ensure that your replicas can keep up with the constant torrent of data
    - Replication replay is single threaded, this can have a huge impact on lagging
- You don't really get your nightly maintenance window
  - Need to be careful with backups
  - Maintenance jobs might need more planning

# Loading Data - When Things Go Wrong



## Loading Data - When Things Go Wrong

- Devices should skew times and back off when things go wrong
  - Can be very easy to trigger congestive collapse
    - Only needs a minor trigger
  - Don't forget this is more about comms, rather than sampling time
- Your devices should still do sensible things without your platform
- Your data loading system should throttle inserts
  - Don't want impact of devices taking your DB out, and thus most of the platform
  - It's probably better to drop data or buffer more than fall flat on your face

**Have head room in your data loading!**



# Managing Data



# Managing Data - Partitioning



## Managing Data - Partitioning

```
CREATE TABLE iot.alhex_reading (  
    device_id    UUID NOT NULL,  
    read_at      TIMESTAMP NOT NULL,  
    temperature  REAL,  
    light        REAL,  
    PRIMARY KEY (device_id, read_at)  
) PARTITION BY RANGE (read_at);
```

## Managing Data - Partitioning

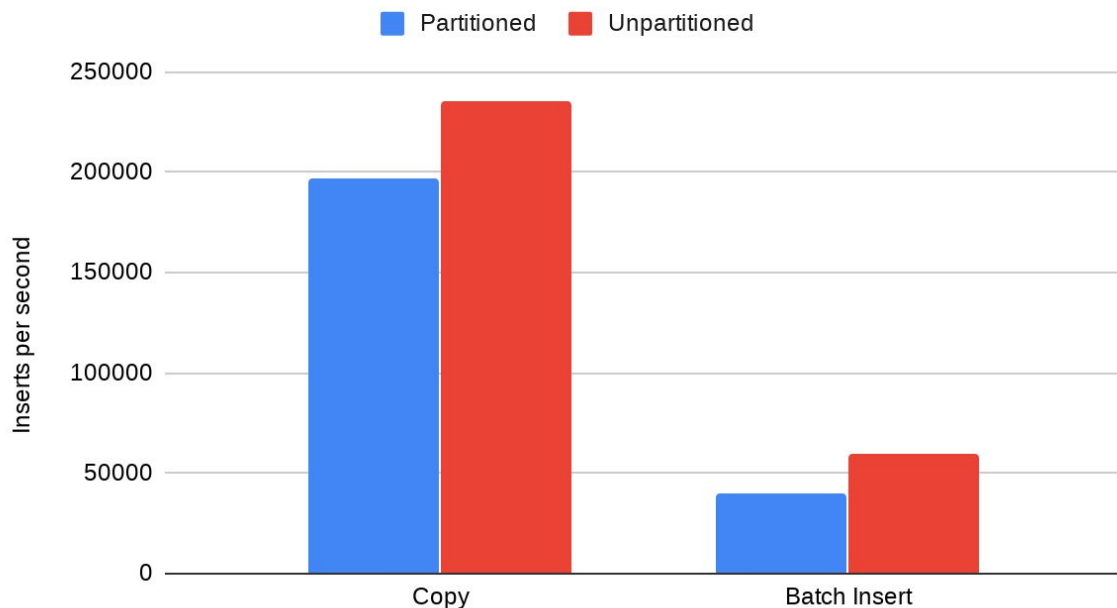
```
CREATE TABLE iot.alhex_reading_201910  
  PARTITION OF iot.alhex_reading  
  FOR VALUES FROM ('2019-10-01') TO ('2019-11-01');
```

...

```
CREATE TABLE iot.alhex_reading_202002  
  PARTITION OF iot.alhex_reading  
  FOR VALUES FROM ('2020-02-01') TO ('2020-03-01');
```

# Managing Data - Partition Loading Performance

Loading Performance (Partitioning)



- Insert into partition parent table
- Inserts need to be directed to the correct partition
- This has a slight performance drop

## Managing Data - Partition Retention

```
ALTER TABLE iot.alhex_reading_201910  
    DETACH PARTITION iot.alhex_reading;
```

```
-- Archive old partition
```

```
COPY iot.alhex_reading_201910  
    TO 'archive/alhex_reading_201910';
```

```
DROP TABLE iot.alhex_reading_201910;
```

## Managing Data - Tablespaces

```
CREATE TABLESPACE archive  
  LOCATION '/data/slow/archive';
```

```
-- Move old data to our archive tablespace
```

```
ALTER TABLE iot.alhex_reading_201910  
  SET TABLESPACE TO archive;
```

# Managing Data - BRIN





## Managing Data - BRIN

```
CREATE TABLE iot.alhex_reading_history (  
    device_id    UUID NOT NULL,  
    read_at      TIMESTAMP NOT NULL,  
    temperature  REAL,  
    light        REAL  
);
```

```
CREATE INDEX alhex_reading_history_read_at_idx  
ON iot.alhex_reading_brin USING BRIN(read_at);
```

## Managing Data - BRIN

```
-- Relation size: 1321 MB, 23,000,000 rows
```

```
SELECT * FROM iot.alhex_reading_history  
WHERE device_id = 'a3e06bcf-429d-43ff-9e46-55aee2ddd86a'  
AND read_at >= '2019-10-17 07:10:31'  
AND read_at <= '2019-10-18 07:10:31';
```

```
-- Seq Scan: 1239 ms          No Index  
-- BRIN:      148 ms          80 kB Index  
-- BTREE:     0.73 ms         891 MB Index
```

# Processing Data



## Processing Data - Putting Stuff Together

```
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;
```

## Processing Data - Putting Stuff Together

```
SELECT avg(r.kwh), min(r.kwh),  
       max(r.kwh), count(*)  
FROM reading_monthly r  
JOIN meter m ON (m.id = r.meter_id)  
JOIN property p ON (m.property_id = p.id)  
WHERE p.bedrooms = 4  
AND r.month BETWEEN '2019-01-01' AND '2019-03-01'
```

## Processing Data - Presenting Data

```
SELECT r.device_id, t.time, array_agg(r.read_at),
       avg(r.temperature), avg(r.light)
FROM generate_series(
    '2019-10-06 00:00:00'::TIMESTAMP,
    '2019-10-07 00:00:00'::TIMESTAMP, '10 minutes') t(time)
JOIN iot.alhex_reading r
    ON (r.device_id = '26170b53-ae8f-464e-8ca6-2faeff8a4d01'::UUID
        AND r.read_at >= t.time
        AND r.read_at < (t.time + '10 minutes'))
GROUP BY 1, 2
ORDER BY t.time;
```

## Processing Data - Presenting Data

```
SELECT r.device_id, t.time, array_agg(r.read_at),
       avg(r.temperature), avg(r.light)
FROM generate_series(
    '2019-10-06 00:00:00'::TIMESTAMP,
    '2019-10-07 00:00:00'::TIMESTAMP, '10 minutes') t(time)
JOIN iot.alhex_reading r
    ON (r.device_id = '26170b53-ae8f-464e-8ca6-2faeff8a4d01'::UUID
        AND r.read_at >= t.time
        AND r.read_at < (t.time + '10 minutes'))
GROUP BY 1, 2
ORDER BY t.time;
```

# Processing Data - Window Functions





## Processing Data - Counters

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

## Processing Data - Rolling Along

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

## Processing Data - Moving On Up

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

# Processing Data - Mind The Gap!



# Processing Data - Mind The Gap

```
WITH days AS (  
    SELECT t.day::DATE  
    FROM generate_series('2017-01-01'::DATE, '2017-01-15'::DATE, '1 day') t(day)  
) , data AS (  
    SELECT *  
    FROM iot.meter_reading  
    WHERE day >= '2017-01-01'::DATE AND day <= '2017-01-15'::DATE  
)  
SELECT day, coalesce(energy_import_wh, (((next_read - last_read) / (next_read_time - last_read_time)) * (day -  
last_read_time)) + last_read) AS energy_import_wh_interpolated  
FROM (  
    SELECT t.day, d.energy_import_wh,  
        last(d.day) OVER lookback AS last_read_time,  
        last(d.day) OVER lookforward AS next_read_time,  
        last(d.energy_import_wh) OVER lookback AS last_read,  
        last(d.energy_import_wh) OVER lookforward AS next_read  
    FROM days t  
    LEFT JOIN data d ON (t.day = d.day)  
    WINDOW  
        lookback AS (ORDER BY t.day),  
        lookforward AS (ORDER BY t.day DESC)  
) q ORDER BY q.day
```

## Processing Data - Mind The Gap

```
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
);
```

## Processing Data - Mind The Gap

```
WITH days AS (  
    SELECT t.day::DATE  
    FROM generate_series('2017-01-01'::DATE,  
    '2017-01-15'::DATE, '1 day') t(day)  
) , data AS (  
    SELECT *  
    FROM iot.meter_reading  
    WHERE day >= '2017-01-01'::DATE  
    AND    day <= '2017-01-15'::DATE  
)
```

## Processing Data - Mind The Gap

```
SELECT t.day, d.energy,  
       last(d.day)      OVER lookback      AS last_read_time,  
       last(d.day)      OVER lookforward   AS next_read_time,  
       last(d.energy)   OVER lookback      AS last_read,  
       last(d.energy)   OVER lookforward   AS next_read  
FROM days t  
LEFT JOIN data d ON (t.day = d.day)  
WINDOW  
  lookback AS (ORDER BY t.day),  
  lookforward AS (ORDER BY t.day DESC)
```



## Processing Data - Mind The Gap

```
SELECT day,  
       coalesce(energy,  
                (((next_read - last_read)  
                 / (next_read_time - last_read_time))  
                 * (day - last_read_time))  
                + last_read) AS energy_interpolated  
FROM (  
    ... from previous slide ...  
) q  
ORDER BY day
```

# Extensions - TimescaleDB

- TimescaleDB is a PostgreSQL extension for time series data
  - Open Source and Commercial licences
- You can do time series data in PostgreSQL without it
  - Nothing I've covered so far needs TimescaleDB
- But TimescaleDB does offer some pretty cool features and is worth having a look at:
  - Benchmarks - 5.4x faster 10% resources compared with Cassandra
  - Hypertables (partitioning), supports 2d partitioning
  - Some very handy functions for dealing with time series data
  - Continuous Views - Build materialised roll up aggregates in real time

# So Long And Thanks For All The Fish

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