Timescaledb: An introduction and brief overview

Brent Wood
October 2018

Timescaledb is a postgresql extension providing optimisations and enhancements for storing and querying timeseries (sensor) data

Background

What is Postgres(ql)?

What is a Postgres extension?

What does timescaledb do/provide?

How do you install it?

What performance & functionality gains are there?

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

Timescaledb is a postgresql extension providing optimisations and enhancements for storing and querying timeseries (sensor) data

Background

https://medium.com/@aiven_io/timescaledb-101-the-why-what-and-how-9c0eb08a7c0b

Current global trend:

More sensors being deployed – vastly more data to manage/access/retrieve
(most “big data” & cloud based datasets are timeseries of some sort)

Open ended growing databases – billions of records – includes NIWA

Historic data often kept online

RDBMS vs NoSQL

Postgres development/popularity (including Postgis)

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What is Postgres(ql)?

Open Source enterprise level ORDBMS

Ongoing uptake globally

Rapid development, user driven

Major player in cloud database services

Acronym (& standards) compliant (PITR, ACID, SQL)

Wide language support (UDF)

Preferred NIWA RDBMS platform

- successful & proven

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What is a Postgres extension?

http://www.asad.pw/awesome-postgres/#extensions

https://pgxn.org/

https://www.postgresql.org/download/products/6-postgresql-extensions/

External, generally third party, tools **embedded and tightly integrated** with the core database engine.

Postgis is perhaps the best known Postgres extension within NIWA

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What does timescaledb do/provide?

Timescaledb is an open source scalable SQL engine for ingesting, managing and querying large volume timeseries datasets.

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What does timescaledb do/provide?

Timescaledb is an open source scalable SQL engine for ingesting, managing and querying large volume timeseries datasets.

It is implemented as an extension to the Postgresql ORDBMS, building on the advanced Postgres client/server ORDBMS engine.

It provides tools to automatically scale to many billions of records, with additional functions to extract timeseries data.

It does nothing that native Postgresql cannot do, in terms of functionality, but provides tools and approaches that significantly simplify the administration and improve the performance of Postgresql for managing large timeseries datasets.

See: https://www.timescale.com/how-it-works

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

Architecture:

Most timeseries databases can be categorised as one of two types:

wide	(lots of tables/columns breaking up large datasets into subsets)
narrow	(lots of records in a well normalised structure, aka deep)

Wide structures perform well on simple queries, but can get bogged down when you are joining lots together. Adding new sensors typically requires structural modifications (eg: new tables). Databases can have hundreds of (small) tables that users need to interact with.

Narrow structures suit the relational model and conventional indexes, but very large datasets can get problematic (don't scale as well). New sensors generally fit within existing structures. They have few tables and simple structures.

Timescaledb tries to be the best of both worlds: the user sees a simple “deep” structure, but underneath, the database automatically splits the data into subsets, based on the actual data. We get the simplicity of working with a few deep tables, with the performance benefits of the underlying hidden wide structures.

Timescaledb: An introduction and brief overview

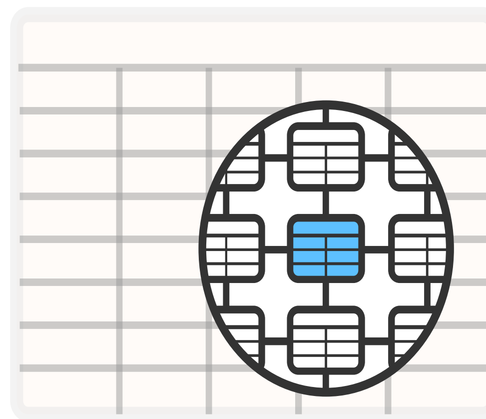
Architecture:

From a user perspective, TimescaleDB exposes what look like singular tables, called hypertables, that are actually an abstraction or a virtual view of many individual tables holding the data, called chunks.

Chunks are created by partitioning the hypertable's data into one or multiple dimensions: All hypertables are partitioned by a time interval, and can additionally be partitioned by a key such as device ID, location, user id, etc.

https://docs.timescale.com/v0.9/introduction/architecture

Hypertable



Chunk

Timescaledb: An introduction and brief overview

How do you install & use it?

1. It requires libraries to be preloaded by the Postgresql server on startup

2. It is installed as an extension in a standard Postgresql (v10+) database

In your postgresql.conf file, add the line:
shared_preload_libraries = 'timescaledb'
Restart postgresql (eg: ***service postgresql restart***)

In your database, run:
CREATE EXTENSION IF NOT EXISTS timescaledb CASCADE;

Timescaledb is now installed & operational in your database. To set up a table of timeseries data to be managed by timeseriesdb, you create the table as usual, then run: ***SELECT create_hypertable('table', 'column');***

There is more if you want to get into details, but this is all that is required.

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What performance & functionality gains are there?

and “how long is a piece of string?” and other short stories...

NIWA maintains a DAS database. This is a Postgresql database storing vessel (mostly Tangaroa) instrument readings at 1 minute intervals, captured by the vessels' “Data Aquisition System”.

It is currently running on an old SLES server on Postgres v8.3.14.
(FYI: v11 has just been released – DAS db is over 100 releases old)

It needs upgrading, so I used this as a test dataset to see if Timescaledb offers any advantages over standard Postgres for NIWA's DAS database...

The main reading table has about 550,000,000 records, so is reasonably substantial. It is currently growing by about 50,000,000 records per year.

The rest of the presentation describes the system & shows some timed SQL commands run on both a native table and a timescaledb hypertable holding the same data.

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What performance & functionality gains are there?

Search one.niwa.co.nz for “DAS database” for information,
https://one.niwa.co.nz/download/attachments/18355008/DAS%20database.pdf

The documents follow the layout used for the MPI research databases managed by NIWA for MPI.

The Timescaledb & Postgres settings have NOT been tuned or experimented with, so any results can probably be improved
– a very simple trial.

The original DAS data contains 60 records with a null timestamp – these cannot be stored in a Timescaledb hypertable – so were dropped from the hypertable.

Simplistic hot and cold timings were usually carried out.

Cold: immediately after a postgresql server restart
Hot: the second time the same query was run sequentially

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What performance & functionality gains are there?

All tests carried out on under Linux Mint v19, Postgresql v10.5, Timescaledb v 0.12, on an i7 laptop with 8Gb memory & a 480Gb SSD.

All SQL commands were run from the Linux shell:
eg: *psql -d das -c "\timing" -c "select count(*) from t_reading;"*

Thus each command ran two SQL statements, the first turned timing on so the elapsed time was reported at the completion of the second command.

t_reading is the native Postgresql table with all records.
ts_reading is the Timeseriesdb hypertable

The new hypertable was populated using the SQL:

```
insert into ts_reading
select * from t_reading
where timer notnull;
```

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What performance & functionality gains are there?

Query: *select count(*) from table;*

Cold native:	104923ms	(1m 45s)
Hot native:	104530ms	(1m 44s)
Cold timescale:	188157ms	(3m 08s)
Hot timescale:	110930ms	(1m 51sec)

8Gb of memory for caching is no real help with a table scan, so there is no real advantage in a hot query over cold for a native table.

Indexes are of little value as every record in the table is processed.

Scanning many chunks & partitions comprising a hypertable adds a fair bit of overhead, but timescaledb/OS optimisations largely negate this for a hot scan.

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What performance & functionality gains are there?

```
Query: select  count(*) as no from table
       where timer >='01/01/2017'::date
         and timer <  '01/01/2018'::date;
```

Cold native:	20318ms	(00m 20s)
Hot native:	05671ms	(00m 05s)
Cold timescale:	04771ms	(00m 05s)
Hot timescale:	02173ms	(00m 02s)

Timescaledb is about 3-4x faster for a simple query with a date range.
A hot query is significantly faster in both cases.

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What performance & functionality gains are there?

Query: *select extract(year from timer) as yr*
count(*) as no from table
where yr=2017
group by yr;

Cold native:	472787ms	(07m 53s)
Hot native:	464911ms	(07m 45s)
Cold timescale:	116057ms	(01m 56s)
Hot timescale:	122641ms	(02m 03s)

Timescaledb is about 4x faster for a simple group by query with a date constraint.
A hot query shows no advantage, and not enough iterations were run
for any detailed conclusions.

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What performance & functionality gains are there?

```
select t1.timer as timer
         t2.value as lat,
         t3.value as lon,
from t_reading t1,
        t_reading t2
where t1.timer >= '01/01/2016'::date
    and t1.timer <  '01/02/2016'::date
    and t1.measurement_key in (231,196,73,31)
    and t2.measurement_key in (232,197,74,32);
```

Cold native:	56755ms	(00m	57s)
Hot native:	55411ms	(00m	55s)
Cold timescale:	52229ms	(00m	52s)
Hot timescale:	51645ms	(00m	52s)

Timescaledb is slightly faster, but much of the work is in creating the join & the measurment_key filter, where Timescaledb is having no impact.

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What performance & functionality gains are there?

Query: *select t1.timer::date as _date,*

```
select t1.timer::date as _date,
       avg(t1.value) as daily_lat,
       avg(t2.value) as daily_lon
  from t_reading t1, t_reading t2
  where t1.timer >= '01/01/2001'::date
    and t1.timer <   '01/02/2001'::date
    and t2.timer = t1.timer
    and t1.measurement_key in (231,186,73,31)
    and t2.measurement_key in (232,187,73,32)
  group by _date
  order by _date;
```

Cold native:	66680ms	(01m 07s)
Hot native:	55412ms	(00m 55s)
Cold timescale:	41499ms	(00m 41s)
Hot timescale:	41694ms	(00m 42s)

Timescaledb (cold) is about 25% faster than native postgres (hot)

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What performance & functionality gains are there?

Query: *select timer::date as date,*

```
select timer::date as date,
       count(*) as no
from t_reading
where extract(doy from timer)=01
group by date
order by date;
```

Cold native:	91359ms	(01m 31s)
Hot native:	91521ms	(01m 32s)
Cold timescale:	99475ms	(01m 39s)
Hot timescale:	99032ms	(01m 39s)

Timescaledb is about 9% slower than native postgres

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What performance & functionality gains are there?

Query:

```
select  date_trunc('day', timer) as day
            count(*) as no
    from t_reading
    where timer >= '01/01/2010'::date
       and timer <= '31/01/2010'::date
    group by day
    order by day;
```

Cold native: 2183ms   (00m 2s)
Hot native: 1434ms   (00m 1s)

Cold timescale: 1134ms(00m 01s)
Hot timescale: 825ms   (00m <1s)

Timescaledb is about 1.5-2x faster than native postgres

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What performance & functionality gains are there?

Query: *select distinct timer*
from t_reading
where (timer >= '2002/12/25'::date and
timer <= '2003/01/15'::date)
and ((measurement_key = 31 or
measurement_key = 73 or
measurement_key = 93 or
measurement_key = 116 or
measurement_key = 196 or
measurement_key = 231) and
value between -44 and -43.7)
and ((measurement_key = 32 or
measurement_key = 74 or
measurement_key = 94 or
measurement_key = 117 or
measurement_key = 196 or
measurement_key = 232) and
value between 184.2 and 184.5);

Are there any records in this time and place...

need to use all lat/lon devices!

Cold native: 1596ms (1.6s)

Cold timescale: 36ms (0.04s)

Timescale is 30x faster

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What performance & functionality gains are there?

Query:
```
select time_bucket('fifteen minutes', timer) as fifteen_min,
        avg(value) as avg_sst
from ts_reading
where extract(year from timer) = 1995
    and measurement_key=130
group by fifteen_min
order by fifteen_min;
```

The *time_bucket()* function only works with timescaledb tables.

It ran in 46932ms (47s) for a year's worth of averaged SST's at 15 minute intervals

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

What performance & functionality gains are there?

Generally timescaledb (note: v 0.12) equals or betters native Postgresql performance.

Functions such as “time_bucket()” are useful extensions.

In cases where the main workload is predicated on time filters, it can be significantly faster (approaching 4x faster – though claims of 20x are made, and in one case it was 30x)

These tests did NOT optimise Postgresql server settings, which is likely to improve both native and timescaledb performance.

They did NOT use native partitioning or clustered indexes, which are likely to improve native postgres performance (but require maintenance).

They did NOT create extra chunks in the hypertable based on instrument, which (theoretically!) will significantly improve Timescaledb performance.

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

Conclusion.

This was a **very** limited trial, but found no issues using Timescaledb.

Installation was very simple, and it worked as promised.

Online documentation was good, providing all the information needed to install and trial Timescaledb.

A table of 550,000,000 records performs quite adequately on a reasonable spec laptop (with both native Postgres and Timescaledb).

It can be much faster and has some useful extensions to Postgres for querying time series data.

I found no reason not use it (unless all your queries require full table scans) if you are managing a timeseries database.

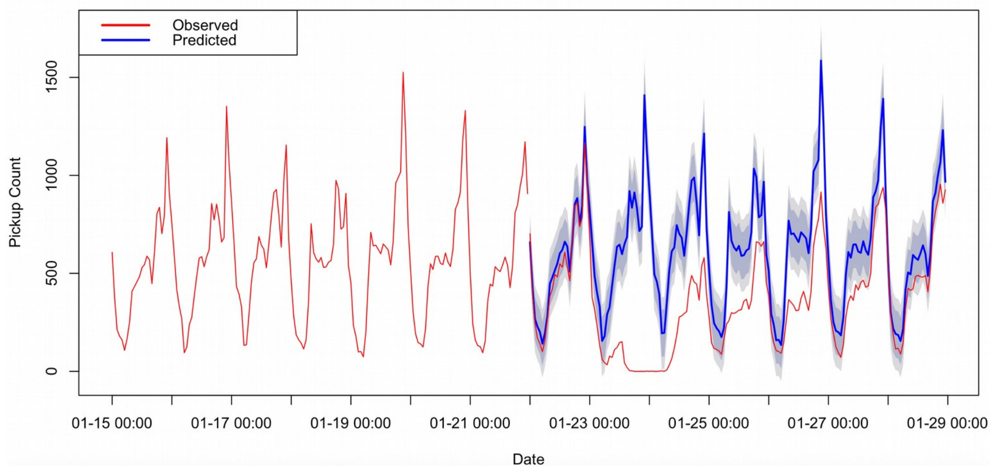
NIWA
Taihoro Nukurangi

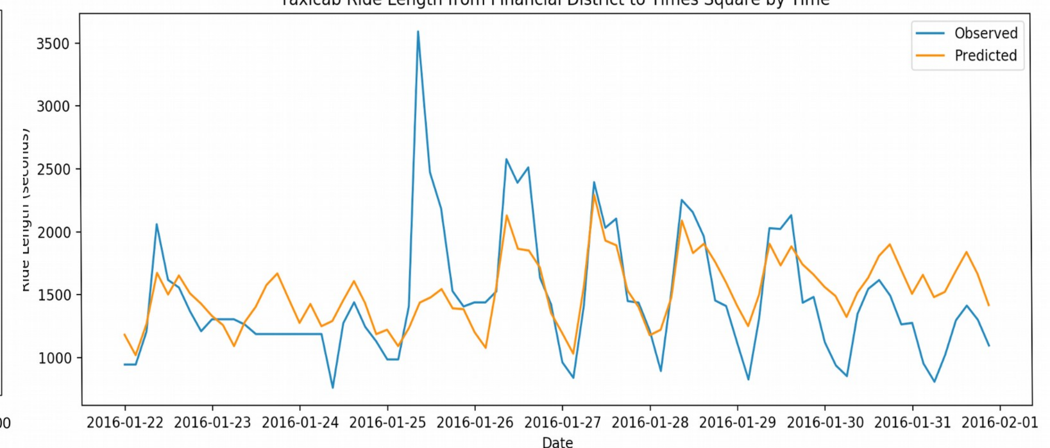
Timescaledb: An introduction and brief overview

Analytics

The Timescaledb web site has an interesting example of timeseries data analytics and forecasting (ARIMA model)...

https://docs.timescale.com/v0.12/tutorials/tutorial-forecasting





Python, R: (data export)

+ Apache, MADlib: in database

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

Crystal ball gazing

Timescaledb is new (just at v1.0) & will continue to improve

The Postgres core dev deam is very interested in Timescale
- useful Timescale functionality will be integrated in core Postgres

Postgres v11 was released this week - on Tuesday 16 Oct (2018)
- this includes significant advances in parallel processing
    - (a single query can split the task between multiple
      “worker processes”)

I'm guessing that this combined with Timescaledb's splitting up of datasets into separate chunks/shards/partitions will provide serious performance gains over the next couple of years

NIWA
Taihoro Nukurangi

Timescaledb: An introduction and brief overview

Thank you for your time, hopefully this was relevant & useful...

Any questions??

NIWA
Taihoro Nukurangi