TimescaleDB Workshop

Authors: | Michael Ward, Casey Smith, Christion Victor Smith

What is TimescaleDB

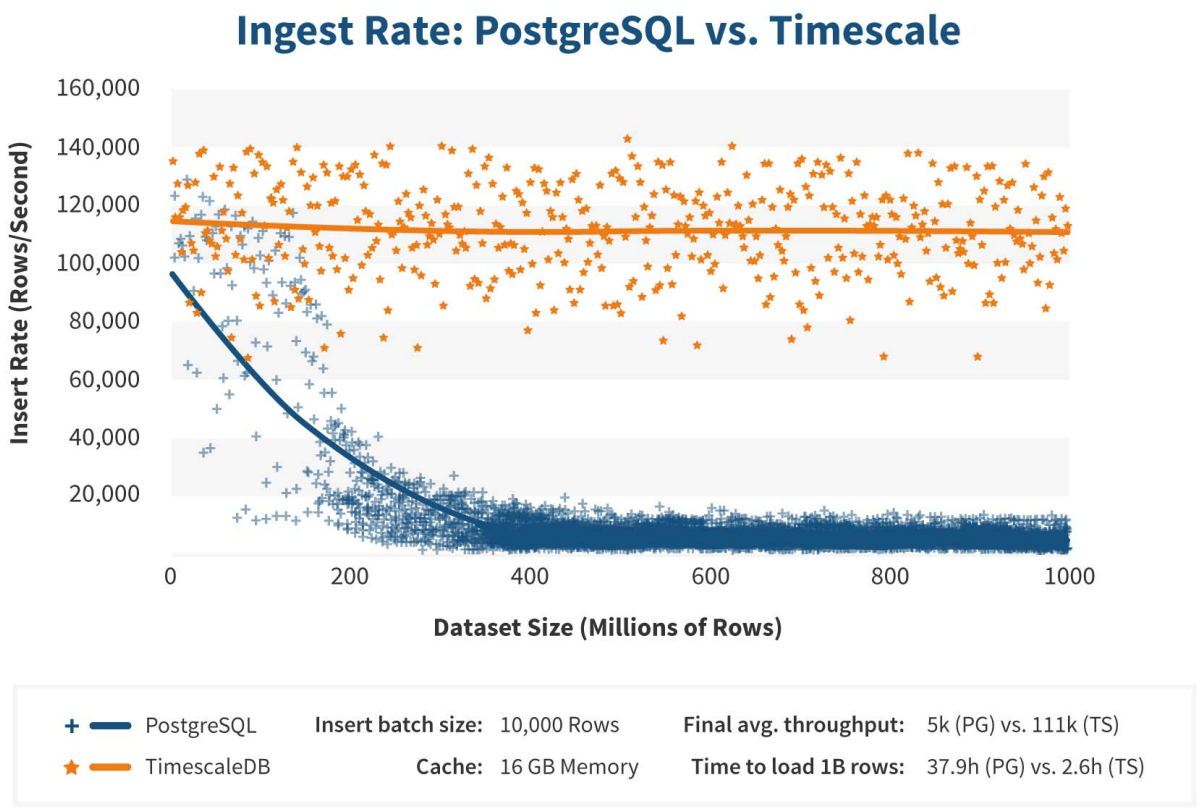
TimescaleDB is an open-source time-series database that is optimized for fast ingest and complex queries. It is engineered up from PostgreSQL and is packaged as an extension.

Time-series data can be described as data that collectively represents how a system, process, or behaviour changes over time. The data records should always have a timestamp.

You can use PostgreSQL to write time-based queries, why use Timescale?

There are a couple of benefits that TimescaleDB has over Postgres, or other traditional relational databases that can store time series data.

- **Scalability** (hence, time-scale) - Normal databases are not designed to handle the scale at which most time-series data accumulates. TimescaleDB is extremely efficient for ingesting and querying large datasets.
- **Query speed** - Simple queries involving a basic scan over an index have similar performance between Postgres and TimescaleDB. Large queries involving time-based GROUP BYs however, perform much faster in TimescaleDB.
- **Usability** - TimescaleDB has built in functions and operations that are common to time-series data analysis, such as data retention policies, continuous queries, time bucketing, and more. We will cover some of the functions and operations present in TimescaleDB in this workshop



(Source: https://docs.timescale.com/latest/introduction/timescaledb-vs-postgres)

What this workshop covers:

So far, we have demonstrated how to set up TimescaleDB using Docker. This workshop will introduce the architecture of TimescaleDB, the functions available in TimescaleDB, and an overview of using time-based queries in conjunction with spatial queries.

Further Uses for TimescaleDB: Topics we will not be covering

This workshop is an introduction to TimescaleDB and time-series data, there are a lot of cool technologies that TimescaleDB integrates with that we will not cover. Some of them are listed here with links if you'd like to learn more on your own.

Data Visualization with Grafana

Grafana is an open source analytics and monitoring solution that is often used to visualize time-series data. It also supports geospatial dashboards. You can use Grafana to visualize metrics and geospatial data stored in TimescaleDB.

(Source: https://www.reddit.com/r/grafana/comments/gepiam/my_vitals_dashboard_3_weeks_of_learningtinkering/)

(Source: https://docs.timescale.com/latest/tutorials/tutorial-grafana-geospatial)

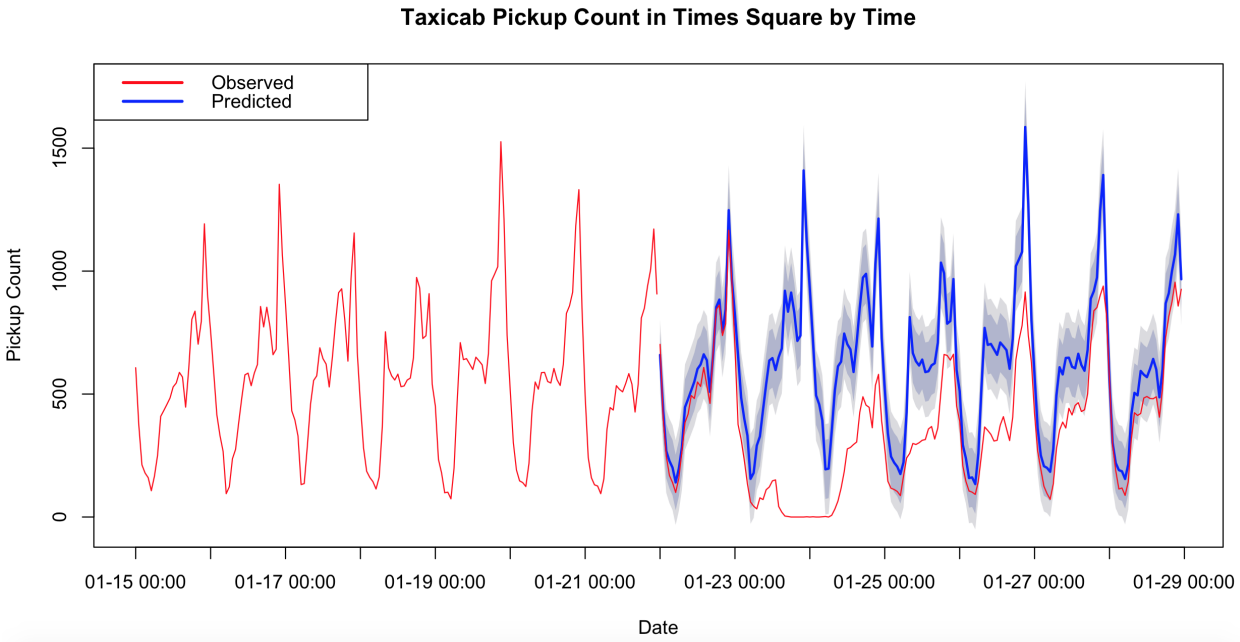
Links:

- https://docs.timescale.com/latest/tutorials/tutorial-grafana
- https://grafana.com/

Time-Series Forecasting

By combining TimescaleDB with various programming languages, you can perform time-series forecasting methods. If you have accumulated thousands or millions of data points over a time period, you can build models to predict the next set of values likely to occur.

TimescaleDB supports integration with Python, R, and Apache MADlib.



(Source: https://docs.timescale.com/latest/tutorials/tutorial-forecasting)

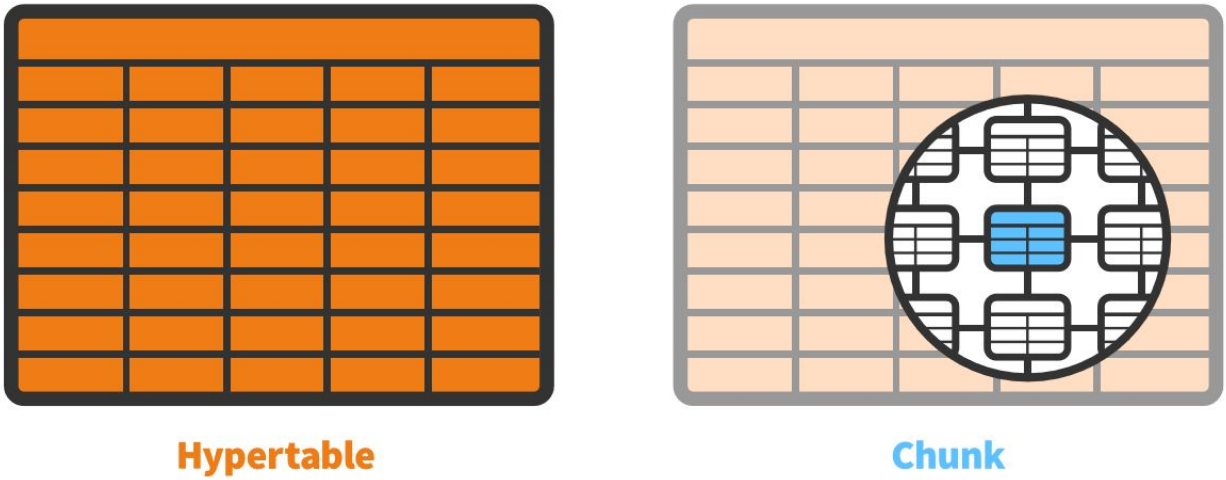
Links:

- https://docs.timescale.com/latest/tutorials/tutorial-forecasting

TimescaleDB architecture

While timescaleDB is implemented as an extension of PostgreSQL, it heavily customizes and transforms key features of PostgreSQL specifically within the query planner, data model, and execution engine.

One large customization to Postgres is the hypertable. The hypertable is an abstraction from a normal table in Postgres. Behind the scenes, hypertables are broken up into many smaller individual tables holding the data, which are called chunks.



(source: https://docs.timescale.com/latest/introduction/architecture)

One cool thing about working in DBeaver with your TimescaleDB database is that you can actually see the hypertable structure by viewing the ERD for your hypertable.

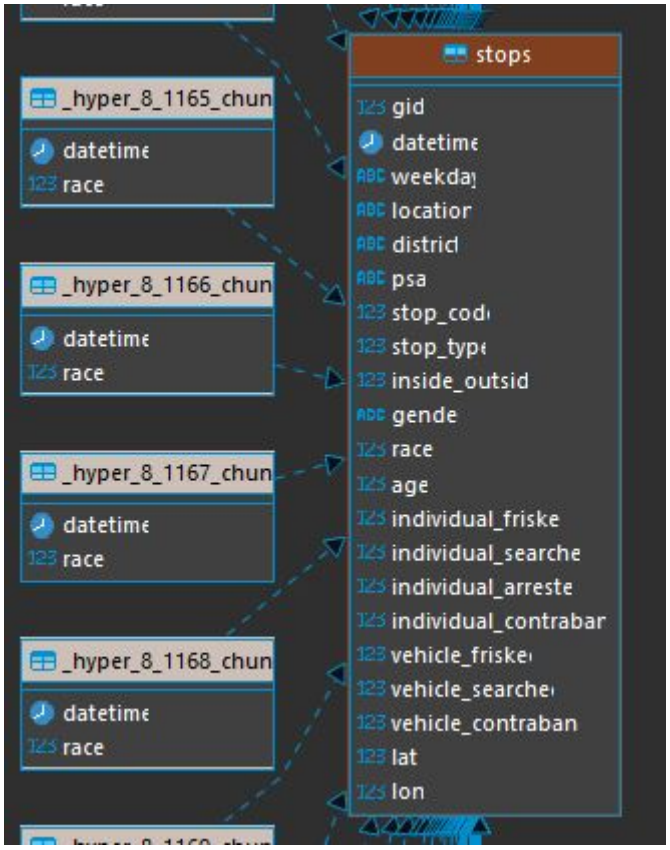


Figure 1: ERD

Hypertables are always partitioned on time, and have the option to partition on additional columns. If you check back on the ERD you can see that the hypertable is partitioned on time and race.

The "Chunks" of the hypertable each correspond to a specific time interval and are non-overlapping, which

allows the query planner to minimize the number of chunks that must be touched to resolve a query. This important structural feature is part of what makes TimescaleDB queries so fast.

Chunks are implemented using a standard database table, and are considered child tables to the parent hypertable.

`create_hypertable()` is just one of the function that TimescaleDB introduces. `create_hypertable()` allows you to convert a regualar table that has a timestamp column to a hypertable.

The arguments for `create_hypertable()` are as follows:

Required Args :

Name	Description
`relation`	Name of table to convert to hypertable
`time_column_name`	Name of column containing time values (primary partition column)

Optional Args :

Name	Description
`partitioning_column`	Name of additional column to partition by. Must also define `number_partitions` argument if used.
`number_partitions`	Number of hash partitions to use for the `partitioning_column`. Must be > 0
`chunk_time_interval`	Interval in event time that each chunk covers. Must be > 0. Default is 7 days for version0.11.0+, previous versions default is 1 month
`create_default_indexes`	Boolean value that determines if default indexes on time/partitioning columns are created. Default is TRUE
`if_not_exists`	Boolean whether to print warning if table is already converted to hypertable. Default is FALSE
`partitioning_func`	Function to use for calculating a value's partition
`associated_schema_name`	Name of the schema for internal hypertable tables. Default is "_timescaledb_internal"
`associated_table_prefix`	Prefix for internal hypertable chunk names. Default is "_hyper"
`migrate_data`	Set to TRUE if you need to migrate existing data from the `relation` table to chunks in new hypertable. Non-empty tables will generate an error without this function. Large tables will take significant time to migrate. Defaults to FALSE
`time_partitioning_func`	Function to convert incompatible primary time column values to compatible ones. The function must be IMMUTABLE
`replication_factor`	If set to 1 or greater, will create a distributed hypertable. Default is NULL. There is also a seperate function called `create_distributed_hypertable`
`data_nodes`	This is the set of data nodes that will be used for this table if it is distributed

(Source: https://docs.timescale.com/latest/api#create_hypertable)

TimescaleDB Functions

This list of functions is not exhaustive, but covers the most used functions introduced to PostgreSQL by TimescaleDB. For an exhaustive list of functions, reference the documentation: https://docs.timescale.com/latest/api#hypertable-management

`time_bucket()` :

A more powerful version of the standard PostgreSQL date_trunc() function. date_trunc() truncates a TIMESTAMP or an INTERVAL value based on a specified date part (e.g. hour, week or month) and returns the truncated timestamp or interval. It accepts arbitrary time intervals as well as optional offsets and returns the bucket start time.

Required Args :

Name	Description
`bucket_width`	A PostgreSQL time interval for how long each bucket is (interval)
`time`	The timestamp to bucket (timestamp/timestamptz/date)

Optional Args :

Name	Description
`offset`	The time interval to offset all buckets by (interval)
`origin`	Buckets are aligned relative to this timestamp (timestamp/timestamptz/date)

`last()` and `first()` :

These functions allow you to get the value of one column as ordered by another. This is commonly used in an aggregation, such as getting the first or last element of that group. For example, last(temperature, time) will return the latest temperature value based on time within a group (e.g., an hour).

Required Args (same for both) :

Name	Description
`value`	A The value to return (anyelement)
`time`	The timestamp to bucket (timestamp/timestamptz/date)

`histogram()` :

This function represents the distribution of a set of values as an array of equal-width buckets. It partitions the dataset into a specified number of buckets (nbuckets) ranging from the inputted min and max values.

Required Args :

Name	Description
`value`	A set of values to partition into a histogram
`min`	The histogram's lower bound used in bucketing (inclusive)
`max`	The histogram's upper bound used in bucketing (exclusive)
`nbuckets`	The integer value for the number of histogram buckets (partitions)

`time_bucket_gapfill()` :

This function essentially allows for the creation of time values that do not have any recorded data. For example, because the stock market is only open for trading on weekdays, there would be no date entries for 4/17/21 and 4/18/21, along with all other dates that fall on weekends. In order to generate a series of time buckets according to some interval across a specified period.

Required Args :

Name	Description
`bucket_width`	A PostgreSQL time interval for how long each bucket is (interval)
`time`	The timestamp to bucket (timestamp/timestamptz/date)

Optional Args :

Name	Description
`start`	The start of the gapfill period (timestamp/timestamptz/date)
`finish`	The end of the gapfill period (timestamp/timestamptz/date)

Analysis with TimescaleDB

Temporal Queries using PostgreSQL functions

Before getting started make sure to set your search path.

```
set search_path = stop_frisk, public;
```

We are going to start by looking at the existing way to do a temporal query using Postgresql.

What's the total number of stops that occured each year for the entire period of 2018- March 2021?

```
select date_trunc('year', datetime) as year, count(*)
from stops
group by year
order by year;
```

What's the total number of stops that occured each month for the entire period of 2018- March 2021?

```
select date_trunc('month', datetime) as month, count(*)
from stops
group by month
order by month;
```

What's the total number of stops that took place each week over the entire period of 2018- March 2021?

```
select date_trunc('week', datetime) as week, count(*)
from stops
group by week
order by week;
```

What's the total number of stops that that took place each hour for the entire period of 2018- March 2021?

```
select date_trunc('hour', datetime) as hour, count(*)
from stops
group by hour
order by hour;
```

Challenge : What's the total number of stops that took place each day in April 2020? Group and the order the result by day.

How many stop occurred for each race type in January 2019?

```
SELECT race, COUNT(*) AS num_stops
FROM stops
WHERE datetime < '2019-02-01 00:00:00'
GROUP BY race
```

```
ORDER BY (num_stops) desc;
```

More detailed table with racial description:

```
SELECT race.description, COUNT(*) AS num_stops,
  RANK () OVER (ORDER BY COUNT(*) DESC) AS race_rank FROM stops
  JOIN race ON race.race = stops.race
  where datetime < '2019-02-01 00:00:00'
  GROUP BY race.description;
```

Time Bucket Queries using TimescaleDB

How many stops took place on May 30th 2019 in five minute intervals?

Without using TimescaleDB:

```
SELECT
  EXTRACT(hour from datetime) as hours,
  trunc(EXTRACT(minute from datetime) / 5)*5 AS five_mins,
  COUNT(*)
FROM stops
WHERE datetime between '2019-05-30 00:00:00' and '2019-05-30 23:59:00'
GROUP BY hours, five_mins;
```

Same query but using TimescaleDB:

```
SELECT time_bucket('5 minute', datetime) AS five_min, count(*)
FROM stops
WHERE datetime between '2019-05-30 00:00:00' and '2019-05-30 23:59:00'
GROUP BY five_min
ORDER BY five_min;
```

Spatial Temporal Queries

Create postgis extension for spatial queries

```
CREATE EXTENSION postgis;
```

Add geometry column to main stops table

```
ALTER TABLE stops ADD COLUMN stops_geom geometry(POINT,6347);
```

Insert newly created geometry into geometry column

```
UPDATE stops SET stops_geom = ST_Transform(ST_SetSRID(ST_MakePoint(lon,lat),4326),6347);
```

Examining Police Stop and Frisk during BLM protest during George Floyd and Walter Wallace Jr Protests

How many stops occured on May 30th to June 6th originated from within 487.68m (4 city blocks) of City Hall, in 30 minute buckets?

Before:

```
SELECT time_bucket('30 minutes', datetime) AS thirty_min, COUNT(*) AS num_stops, stops_geom
FROM stops
WHERE ST_Distance(stops_geom, ST_Transform(ST_SetSRID(ST_MakePoint(-75.16352621534354,
39.953425054848935),4326),6347)) < 487.68
AND datetime between '2020-05-23 00:00:00' and '2020-05-29 23:59:00'
GROUP BY thirty_min, stops_geom
ORDER BY thirty_min;
```

During:

```
SELECT time_bucket('30 minutes', datetime) AS thirty_min, COUNT(*) AS num_stops, stops_geom
FROM stops
WHERE ST_Distance(stops_geom, ST_Transform(ST_SetSRID(ST_MakePoint(-75.16352621534354,
39.953425054848935),4326),6347)) < 487.68
AND datetime between '2020-05-30 00:00:00' and '2020-06-06 23:59:00'
GROUP BY thirty_min, stops_geom
ORDER BY thirty_min;
```

After:

```
SELECT time_bucket('30 minutes', datetime) AS thirty_min, COUNT(*) AS num_stops, stops_geom
FROM stops
WHERE ST_Distance(stops_geom, ST_Transform(ST_SetSRID(ST_MakePoint(-75.16352621534354,
39.953425054848935),4326),6347)) < 487.68
AND datetime between '2020-06-07 00:00:00' and '2020-06-14 23:59:00'
GROUP BY thirty_min, stops_geom
ORDER BY thirty_min;
```

How many stops occured on May 30th originated from within 1609.344m (1 mile) of Art Museum, in 30 minute buckets?

Week before:

```
SELECT time_bucket('30 minutes', datetime) AS thirty_min, COUNT(*) AS num_stops, stops_geom
FROM stops
WHERE ST_Distance(stops_geom, ST_Transform(ST_SetSRID(ST_MakePoint(-75.1810519306871,
39.966227529457264),4326),6347)) < 1609.344
AND datetime between '2020-05-23 00:00:00' and '2020-05-29 23:59:00'
GROUP BY thirty_min, stops_geom
ORDER BY thirty_min;
```

During:

```
SELECT time_bucket('30 minutes', datetime) AS thirty_min, COUNT(*) AS num_stops, stops_geom
FROM stops
WHERE ST_Distance(stops_geom, ST_Transform(ST_SetSRID(ST_MakePoint(-75.1810519306871,
39.966227529457264),4326),6347)) < 1609.344
AND datetime between '2020-05-30 00:00:00' and '2020-06-06 23:59:00'
GROUP BY thirty_min, stops_geom
ORDER BY thirty_min;
```

Week after:

```
SELECT time_bucket('30 minutes', datetime) AS thirty_min, COUNT(*) AS num_stops, stops_geom
FROM stops
WHERE ST_Distance(stops_geom, ST_Transform(ST_SetSRID(ST_MakePoint(-75.1810519306871,
39.966227529457264),4326),6347)) < 1609.344
AND datetime between '2020-06-07 00:00:00' and '2020-06-14 23:59:00'
GROUP BY thirty_min, stops_geom
ORDER BY thirty_min;
```

How many stop occured between oct 26th and Nov 1st 2020? (Walter Wallace Jr. protests)

Week Before

```
SELECT time_bucket('12 hours', datetime) AS twelvehours, COUNT(*) AS num_stops, stops_geom
FROM stops
WHERE ST_Distance(stops_geom, ST_Transform(ST_SetSRID(ST_MakePoint(-75.221568086892,
```

```
39.97514562111139),4326),6347)) < 1609.344
AND datetime between '2020-10-19 00:00:00' and '2020-10-25 23:59:00'
GROUP BY twelvehours, stops_geom
ORDER BY twelvehours;
```

During:

```
create temp table wwduring as
SELECT time_bucket('12 hours', datetime) AS twelvehours, COUNT(*) AS num_stops, stops_geom
FROM stops
WHERE ST_Distance(stops_geom, ST_Transform(ST_SetSRID(ST_MakePoint(-75.221568086892,
39.97514562111139),4326),6347)) < 1609.344
AND datetime between '2020-10-26 00:00:00' and '2020-10-30 23:59:00'
GROUP BY twelvehours, stops_geom
ORDER BY twelvehours;
```

Week after:

```
SELECT time_bucket('12 hours', datetime) AS twelvehours, COUNT(*) AS num_stops, stops_geom
FROM stops
WHERE ST_Distance(stops_geom, ST_Transform(ST_SetSRID(ST_MakePoint(-75.221568086892,
39.97514562111139),4326),6347)) < 1609.344
AND datetime between '2020-11-02 00:00:00' and '2020-11-08 23:59:00'
GROUP BY twelvehours, stops_geom
ORDER BY twelvehours;
```

Challenge:

Examine police pedestrian stops that occur every 12 hours at Temple University during the fall, spring and summer semesters during the 2018-2019 academic year?

- coordinates: 39.98135053704874, -75.15586634603065
- distance: 1609.34 (.5 Miles)
- fall: Aug 27 - Dec 19
- Spring Jan 14 - May 8th
- Summer May 13 - Aug 11