TimescaleDB

Step 2: Connect to TimescaleDB

Connect using `psql`:

```
psql -h localhost -p 6543 -U postgres
# When prompted, enter password: password
```

You should see the PostgreSQL prompt. Verify TimescaleDB is installed:

```
SELECT extname, extversion FROM pg_extension WHERE extname = 'timescaledb';
```

Expected output:

```
   extname   | extversion
-------------+------------
 timescaledb | 2.x.x
```

Prefer a GUI? If you'd rather use a graphical tool instead of the command line, you can download pgAdmin and connect to TimescaleDB using the same connection details (host: `localhost`, port: `6543`, user: `postgres`, password: `password`).

Step 3: Create Your First Hypertable

Let's create a hypertable for IoT sensor data with columnstore enabled:

```
-- Create a hypertable with automatic columnstore
CREATE TABLE sensor_data (
    time TIMESTAMPTZ NOT NULL,
    sensor_id TEXT NOT NULL,
    temperature DOUBLE PRECISION,
    humidity DOUBLE PRECISION,
    pressure DOUBLE PRECISION
) WITH (
    tsdb.hypertable
);
```

`tsdb.hypertable` - Converts this into a TimescaleDB hypertable

See more:

- About hypertables
- API reference
- About columnstore
- Enable columnstore manually
- API reference

Step 4: Insert Sample Data

Let's add some sample sensor readings:

```
-- Enable timing to see time to execute queries
\timing on

-- Insert sample data for multiple sensors
-- SET timescaledb.enable_direct_compress_insert = on to insert data directly to the columns
SET timescaledb.enable_direct_compress_insert = on;
INSERT INTO sensor_data (time, sensor_id, temperature, humidity, pressure)
SELECT
    time,
    'sensor_' || ((random() * 9)::int + 1),
    20 + (random() * 15),
    40 + (random() * 30),
    1000 + (random() * 50)
FROM generate_series(
    NOW() - INTERVAL '90 days',
    NOW(),
    INTERVAL '1 seconds'
) AS time;

-- Once data is inserted into the columnstore we optimize the order and structure
-- this compacts and orders the data in the chunks for optimal query performance and compres
DO $$
DECLARE ch TEXT;
BEGIN
    FOR ch IN SELECT show_chunks('sensor_data') LOOP
        CALL convert_to_columnstore(ch, recompress := true);
    END LOOP;
END $$;
```

This generates ~7,776,001 readings across 10 sensors over the past 90 days.

Verify the data was inserted:

```
SELECT COUNT(*) FROM sensor_data;
```

Step 5: Run Your First Analytical Queries

Now let's run some analytical queries that showcase TimescaleDB's performance:

```
-- Enable query timing to see performance
\timing on

-- Query 1: Average readings per sensor over the last 7 days
SELECT
    sensor_id,
    COUNT(*) as readings,
    ROUND(AVG(temperature)::numeric, 2) as avg_temp,
    ROUND(AVG(humidity)::numeric, 2) as avg_humidity,
```

```
    ROUND(AVG(pressure)::numeric, 2) as avg_pressure
FROM sensor_data
WHERE time > NOW() - INTERVAL '7 days'
GROUP BY sensor_id
ORDER BY sensor_id;

-- Query 2: Hourly averages using time_bucket
-- Time buckets enable you to aggregate data in hypertables by time interval and calculate s
SELECT
    time_bucket('1 hour', time) AS hour,
    sensor_id,
    ROUND(AVG(temperature)::numeric, 2) as avg_temp,
    ROUND(AVG(humidity)::numeric, 2) as avg_humidity
FROM sensor_data
WHERE time > NOW() - INTERVAL '24 hours'
GROUP BY hour, sensor_id
ORDER BY hour DESC, sensor_id
LIMIT 20;

-- Query 3: Daily statistics across all sensors
SELECT
    time_bucket('1 day', time) AS day,
    COUNT(*) as total_readings,
    ROUND(AVG(temperature)::numeric, 2) as avg_temp,
    ROUND(MIN(temperature)::numeric, 2) as min_temp,
    ROUND(MAX(temperature)::numeric, 2) as max_temp
FROM sensor_data
GROUP BY day
ORDER BY day DESC
LIMIT 10;

-- Query 4: Latest reading for each sensor
-- Highlights the value of Skipscan executing in under 100ms without skipscan it takes over
SELECT DISTINCT ON (sensor_id)
    sensor_id,
    time,
    ROUND(temperature::numeric, 2) as temperature,
    ROUND(humidity::numeric, 2) as humidity,
    ROUND(pressure::numeric, 2) as pressure
FROM sensor_data
ORDER BY sensor_id, time DESC;
```

Notice how fast these analytical queries run, even with aggregations across millions of rows. This is the power of TimescaleDB's columnstore.

What's Happening Behind the Scenes?

TimescaleDB automatically: - **Partitions your data** into time-based chunks for efficient querying - **Write directly to columnstore** using columnar storage (90%+ compression typical) and faster vectorized queries - **Optimizes queries** by only scanning relevant time ranges and columns - **Enables time_bucket()** - a powerful function for time-series aggregation

See more:

- Query data
- Write data
- About time buckets
- API reference
- All TimescaleDB features

Next Steps

Now that you've got the basics, explore more:

Create Continuous Aggregates

Continuous aggregates make real-time analytics run faster on very large datasets. They continuously and incrementally refresh a query in the background, so that when you run such query, only the data that has changed needs to be computed, not the entire dataset. This is what makes them different from regular PostgreSQL materialized views, which cannot be incrementally materialized and have to be rebuilt from scratch every time you want to refresh them.

Let's create a continuous aggregate for hourly sensor statistics:

Step 1: Create the Continuous Aggregate

```
CREATE MATERIALIZED VIEW sensor_data_hourly
WITH (timescaledb.continuous) AS
SELECT
    time_bucket('1 hour', time) AS hour,
    sensor_id,
    AVG(temperature) AS avg_temp,
    AVG(humidity) AS avg_humidity,
    AVG(pressure) AS avg_pressure,
    MIN(temperature) AS min_temp,
    MAX(temperature) AS max_temp,
    COUNT(*) AS reading_count
FROM sensor_data
GROUP BY hour, sensor_id;
```

This creates a materialized view that pre-aggregates your sensor data into hourly buckets. The view is automatically populated with existing data.

Step 2: Add a Refresh Policy To keep the continuous aggregate up-to-date as new data arrives, add a refresh policy:

```
SELECT add_continuous_aggregate_policy(
    'sensor_data_hourly',
    start_offset => INTERVAL '3 hours',
    end_offset => INTERVAL '1 hour',
    schedule_interval => INTERVAL '1 hour'
);
```

This policy: - Refreshes the continuous aggregate every hour - Processes data from 3 hours ago up to 1 hour ago (leaving the most recent hour for real-time queries) - Only processes new or changed data incrementally

Step 3: Query the Continuous Aggregate Now you can query the pre-aggregated data for much faster results:

```
-- Get hourly averages for the last 24 hours
SELECT
    hour,
    sensor_id,
    ROUND(avg_temp::numeric, 2) AS avg_temp,
    ROUND(avg_humidity::numeric, 2) AS avg_humidity,
    reading_count
FROM sensor_data_hourly
WHERE hour > NOW() - INTERVAL '24 hours'
ORDER BY hour DESC, sensor_id
LIMIT 50;
```

Benefits of Continuous Aggregates

- **Faster queries**: Pre-aggregated data means queries run in milliseconds instead of seconds
- **Incremental refresh**: Only new/changed data is processed, not the entire dataset
- **Automatic updates**: The refresh policy keeps your aggregates current without manual intervention
- **Real-time option**: You can enable real-time aggregation to combine materialized and raw data

Try It Yourself Compare the performance difference:

```
-- Query the raw hypertable (slower on large datasets)
\timing on
SELECT
    time_bucket('1 hour', time) AS hour,
    AVG(temperature) AS avg_temp
FROM sensor_data
```

```
WHERE time > NOW() - INTERVAL '60 days'
GROUP BY hour
ORDER BY hour DESC
LIMIT 24;

-- Query the continuous aggregate (much faster)
SELECT
    hour,
    avg_temp
FROM sensor_data_hourly
WHERE hour > NOW() - INTERVAL '60 days'
ORDER BY hour DESC
LIMIT 24;
```

Notice how the continuous aggregate query is significantly faster, especially as your dataset grows!

See more:

- About continuous aggregates
- API reference
- TimescaleDB Documentation
- Time-series Best Practices
- Continuous Aggregates

Examples

Learn TimescaleDB with complete, standalone examples using real-world datasets. Each example includes sample data and analytical queries.

- **NYC Taxi Data** - Transportation and location-based analytics
- **Financial Market Data** - Trading and market data analysis
- **Application Events** - Event logging with UUIDv7

Or try some of our workshops - **AI Workshop: EV Charging Station Analysis** - Integrate PostgreSQL with AI capabilities for managing and analyzing EV charging station data - **Time-Series Workshop: Financial Data Analysis** - Work with cryptocurrency tick data, create candlestick charts

Documentation associated with Apache licensed "community edition" of TimescaleDB provided for convenience and built from Apache licensed source code available here.