Module 2 — Modeling Time-Series Data

This is **Module 2** of the TimescaleDB practical course. It continues from Module 1 (installation & first hypertable). If you completed Module 1, you already have the Docker stack and a simple `sensor_data` example to build on.

Module 2 explains time-series schema design (narrow vs wide), how TimescaleDB partitions by time (hypertables) and gives a hands-on lab that creates hypertables, inserts synthetic data (CPU, temperature, stock prices), then compares performance vs a regular Postgres table. All commands are written for the TimescaleDB image used in Module 1 (TimescaleDB on Postgres 16) and for modern psql / Docker Compose. Follow each step exactly; copy/paste commands into your terminal.

1. Narrow-table vs. wide-table schema design — concepts and examples

Key idea

- **Narrow (tall) table**: one row per timestamp + metric. Good for many metric types and flexible schemas (adds new metrics without altering table); often smaller row width and easier to index by metric/labels.
- **Wide table**: one row per timestamp with many metric columns (cpu, mem, temp, pressure, ...). Good when you always record the same set of metrics at the same timestamp; queries that read many metrics at once are efficient (fewer rows scanned), but schema changes require `ALTER TABLE` and nulls may waste space.

Tradeoffs

- Narrow is flexible and fits tag/labelged metrics well; wide can be faster for single-row-per-timestamp access and reduces row count.
- TimescaleDB (hypertables) supports either pattern; choose by how your producers ingest data and how you query it.

Example schemas

Narrow (tall) schema — one measurement per row:

```
CREATE TABLE metrics_narrow (
  time       TIMESTAMPTZ NOT NULL,
```

```
  device_id   TEXT,     -- or sensor_id INT
  metric      TEXT,     -- e.g., 'cpu', 'temperature', 'stock_price'
  value       DOUBLE PRECISION,
  tags        JSONB     -- optional free-form tags
);
```

Wide schema — many metrics in same row:

```
CREATE TABLE metrics_wide (
  time        TIMESTAMPTZ NOT NULL,
  device_id   TEXT,
  cpu_percent DOUBLE PRECISION,
  temperature DOUBLE PRECISION,
  mem_mb      DOUBLE PRECISION,
  stock_price DOUBLE PRECISION
);
```

In both styles, TimescaleDB hypertables will partition by `time` for efficient time-based queries. If you have a cardinality dimension (sensor/device) that you want to slice by, consider a multi-column partition (time + space via `partitioning_column`) or create appropriate indexes.

2. Partitioning by time (hypertables in TimescaleDB)

TimescaleDB uses **hypertables** as a logical table that is partitioned into many smaller physical chunks along the time dimension (and optionally by a space dimension such as `device_id`). You write SQL against the hypertable and TimescaleDB manages chunks.

Create a hypertable

```
-- From psql (connected to metricsdb)
CREATE TABLE sensor_readings (
  time        TIMESTAMPTZ NOT NULL,
  device_id   INT,
  metric      TEXT,
  value       DOUBLE PRECISION
);

SELECT create_hypertable('sensor_readings', 'time', chunk_time_interval => INTERVAL '1
day');
```

`chunk_time_interval` should be chosen based on ingestion rate and query patterns (1 hour..1 week typical). For low-volume labs `1 day` is fine.

Optional: space partitioning (hypertable with a space column)

```
SELECT create_hypertable(
  'sensor_readings',
  'time',
  partitioning_column => 'device_id',
  number_partitions => 8,
  chunk_time_interval => INTERVAL '1 day'
);
```

This creates both time and space partitioning which helps when you have many distinct `device_id` values and you filter by that column often.

Hands-on lab (easy, step-by-step)

> All commands assume you run them on the machine hosting Docker Compose from Module 1 and use the same container names:

- TimescaleDB container: `timescaledb`
- Postgres user: `admin`
- Database: `metricsdb`

If you use different names/credentials, substitute accordingly. Commands use `docker exec` to run psql inside the container.

A. Create hypertable for IoT sensor data

Connect and create narrow hypertable `sensor_readings`:

```
docker exec -i timescaledb psql -U admin -d metricsdb -v ON_ERROR_STOP=1 <<'SQL'
-- create narrow table for different metric types
CREATE TABLE IF NOT EXISTS sensor_readings (
  time        TIMESTAMPTZ NOT NULL,
  device_id   INT,
  metric      TEXT NOT NULL,
  value       DOUBLE PRECISION,
```

```
  PRIMARY KEY (time, device_id, metric)
);

-- create hypertable (idempotent if already created)
SELECT create_hypertable('sensor_readings', 'time', chunk_time_interval => INTERVAL '1
day');
-- create an index for common queries
CREATE INDEX IF NOT EXISTS idx_sensor_readings_device_metric_time ON
sensor_readings (device_id, metric, time DESC);
SQL
```

(Optional) Create a wide-style hypertable for comparison:

```
docker exec -i timescaledb psql -U admin -d metricsdb -v ON_ERROR_STOP=1 <<'SQL'
CREATE TABLE IF NOT EXISTS sensor_wide (
  time        TIMESTAMPTZ NOT NULL,
  device_id   INT,
  cpu_percent DOUBLE PRECISION,
  temperature DOUBLE PRECISION,
  stock_price DOUBLE PRECISION,
  PRIMARY KEY (time, device_id)
);

SELECT create_hypertable('sensor_wide', 'time', chunk_time_interval => INTERVAL '1 day');
CREATE INDEX IF NOT EXISTS idx_sensor_wide_device_time ON sensor_wide (device_id,
time DESC);
SQL
```

B. Insert synthetic data

We will generate:

- CPU usage (0–100%)
- Temperature (Celsius)
- Stock price (simple random walk)

We will populate both `sensor_readings` (narrow) and `sensor_wide` (wide) with the same underlying logical events so we can compare queries.

Important: for speed we use `INSERT ... SELECT` with `generate_series`. Adjust `ROW_COUNT` if your machine is small. Example below inserts 200,000 logical timestamps for 5 devices → 1,000,000 rows for the narrow table (because narrow stores each metric as its own row). Change `ROW_COUNT` as needed.

Run the following script (single command block):

```
docker exec -i timescaledb psql -U admin -d metricsdb -v ON_ERROR_STOP=1 <<'SQL'
-- Parameters
-- rows_per_device = number of timestamps per device (e.g., 200_000)
-- devices = number of device_ids
-- total narrow rows will be rows_per_device * devices * metrics_per_timestamp

-- For this lab we use smaller default values. Increase for heavier benchmarking.
\set rows_per_device 200000
\set devices 5

-- Clean existing test data (be careful in production)
TRUNCATE sensor_readings, sensor_wide;

-- Insert into sensor_wide (one row per timestamp per device, storing three metrics)
INSERT INTO sensor_wide (time, device_id, cpu_percent, temperature, stock_price)
SELECT
  NOW() - ((:rows_per_device * :devices) - gs) * INTERVAL '1 second' as time,
  ((gs - 1) % :devices) + 1 as device_id,
  (random() * 50 + 10)::double precision as cpu_percent,          -- 10..60%
  (random() * 10 + 15)::double precision as temperature,          -- 15..25 C
  (100.0 + (random()-0.5) * 5.0)::double precision as stock_price
FROM generate_series(1, (:rows_per_device * :devices)) gs;

-- Insert into sensor_readings (narrow): one row per metric per timestamp
-- We'll unpivot the wide results into narrow form efficiently
INSERT INTO sensor_readings (time, device_id, metric, value)
SELECT time, device_id, 'cpu' as metric, cpu_percent::double precision FROM sensor_wide
UNION ALL
SELECT time, device_id, 'temperature' as metric, temperature::double precision FROM
sensor_wide
UNION ALL
SELECT time, device_id, 'stock_price' as metric, stock_price::double precision FROM
sensor_wide;

-- quick counts
SELECT 'sensor_wide' AS table_name, count(*) AS rows FROM sensor_wide;
SELECT 'sensor_readings' AS table_name, count(*) AS rows FROM sensor_readings;
SQL
```

Notes:

- `generate_series` creates a stream of timestamps spaced one second apart backwards from NOW; adjust spacing for different time ranges (e.g., minutes instead of seconds).
- `sensor_readings` will have 3 × rows of `sensor_wide` because each wide row becomes 3 narrow rows.

C. Example queries you will run during comparison

Recent range scan — last 1 hour for device 1, CPU metric (narrow table):

```
\timing on
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT time, value
FROM sensor_readings
WHERE device_id = 1 AND metric = 'cpu' AND time > NOW() - INTERVAL '1 hour'
ORDER BY time DESC
LIMIT 100;
```

Same range scan against the wide table (reading cpu_percent):

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT time, cpu_percent
FROM sensor_wide
WHERE device_id = 1 AND time > NOW() - INTERVAL '1 hour'
ORDER BY time DESC
LIMIT 100;
```

Aggregation (downsample) — average CPU per minute over last 6 hours:

Narrow:

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT time_bucket('1 minute', time) AS minute, avg(value) AS avg_cpu
FROM sensor_readings
WHERE metric = 'cpu' AND time > NOW() - INTERVAL '6 hours'
GROUP BY minute
ORDER BY minute DESC
LIMIT 100;
```

Wide:

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT time_bucket('1 minute', time) AS minute, avg(cpu_percent)::double precision AS
avg_cpu
```

```
FROM sensor_wide
WHERE time > NOW() - INTERVAL '6 hours'
GROUP BY minute
ORDER BY minute DESC
LIMIT 100;
```

Point lookup — latest value per device for CPU:

Narrow (fast using index on device+metric+time):

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT DISTINCT ON (device_id) device_id, time, value
FROM sensor_readings
WHERE metric = 'cpu'
ORDER BY device_id, time DESC;
```

Wide:

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT DISTINCT ON (device_id) device_id, time, cpu_percent
FROM sensor_wide
ORDER BY device_id, time DESC;
```

D. Compare performance with a regular Postgres table (step-by-step)

We will:

1. Create a regular Postgres table (`sensor_plain`) — not a hypertable.
2. Populate it with the same data as `sensor_readings` and run identical queries.
3. Compare `EXPLAIN ANALYZE` results and wall time. Use `psql` with `\timing` on.

1. Create a plain table and populate it

```
docker exec -i timescaledb psql -U admin -d metricsdb -v ON_ERROR_STOP=1 <<'SQL'
-- Create a plain Postgres table with same structure as sensor_readings
DROP TABLE IF EXISTS sensor_plain;
CREATE TABLE sensor_plain (
  time        TIMESTAMPTZ NOT NULL,
  device_id   INT,
  metric      TEXT,
  value       DOUBLE PRECISION
);
```

```
-- Populate sensor_plain from sensor_readings (this duplicates the data)
INSERT INTO sensor_plain (time, device_id, metric, value)
SELECT time, device_id, metric, value FROM sensor_readings;

-- Add the same index used earlier for fair comparison
CREATE INDEX idx_sensor_plain_device_metric_time ON sensor_plain (device_id, metric,
time DESC);

-- Analyze to update planner statistics
ANALYZE sensor_plain;
ANALYZE sensor_readings;
ANALYZE sensor_wide;
SQL
```

2. Run the same queries and capture EXPLAIN ANALYZE

Run the queries from section C against:

- `sensor_readings` (hypertable)
- `sensor_wide` (wide hypertable)
- `sensor_plain` (regular Postgres table)

You can run these commands sequentially to see timings:

```
# Example: range scan on hypertable (narrow)
docker exec -i timescaledb psql -U admin -d metricsdb -c "\timing on" -c "EXPLAIN (ANALYZE,
BUFFERS) SELECT time, value FROM sensor_readings WHERE device_id = 1 AND metric =
'cpu' AND time > NOW() - INTERVAL '1 hour' ORDER BY time DESC LIMIT 100;"

# Same on plain table
docker exec -i timescaledb psql -U admin -d metricsdb -c "EXPLAIN (ANALYZE, BUFFERS)
SELECT time, value FROM sensor_plain WHERE device_id = 1 AND metric = 'cpu' AND time >
NOW() - INTERVAL '1 hour' ORDER BY time DESC LIMIT 100;"
```

3. Measure aggregation performance

```
# Narrow hypertable downsample
docker exec -i timescaledb psql -U admin -d metricsdb -c "EXPLAIN (ANALYZE, BUFFERS)
SELECT time_bucket('1 minute', time) AS minute, avg(value) AS avg_cpu FROM
sensor_readings WHERE metric = 'cpu' AND time > NOW() - INTERVAL '6 hours' GROUP BY
minute ORDER BY minute DESC LIMIT 100;"

# Plain table downsample
docker exec -i timescaledb psql -U admin -d metricsdb -c "EXPLAIN (ANALYZE, BUFFERS)
SELECT time_bucket('1 minute', time) AS minute, avg(value) AS avg_cpu FROM sensor_plain
```

```
WHERE metric = 'cpu' AND time > NOW() - INTERVAL '6 hours' GROUP BY minute ORDER BY minute DESC LIMIT 100;"
```

Expected results and interpretation

- For time-ranged queries and aggregations over a time interval, the hypertable (`sensor_readings`) should be faster due to chunk pruning (only relevant chunks scanned) and TimescaleDB optimizations (time_bucket, parallelization depending on settings).
- The plain table (`sensor_plain`) will scan more rows or require scanning large ranges and may be slower and use more I/O.
- `EXPLAIN (ANALYZE, BUFFERS)` shows actual execution time and buffer reads — compare `actual time` and `shared read`/`shared hit` numbers to judge I/O vs memory.

Quick benchmark script (run after you populated data)

This minimal script runs three representative queries and prints timings for each table. Copy and run it from your shell (create **compare_perf.sh**) (on the host with Docker):

```
#!/usr/bin/env bash
set -euo pipefail

PG="docker exec -i timescaledb psql -U admin -d metricsdb -q -A -t -c"

echo "=== Range query: last 1 hour, device 1, cpu metric ==="
$PG "\timing on" -c "EXPLAIN (ANALYZE, BUFFERS) SELECT time, value FROM sensor_readings WHERE device_id = 1 AND metric = 'cpu' AND time > NOW() - INTERVAL '1 hour' ORDER BY time DESC LIMIT 100;"
$PG "EXPLAIN (ANALYZE, BUFFERS) SELECT time, value FROM sensor_plain WHERE device_id = 1 AND metric = 'cpu' AND time > NOW() - INTERVAL '1 hour' ORDER BY time DESC LIMIT 100;"
$PG "EXPLAIN (ANALYZE, BUFFERS) SELECT time, cpu_percent FROM sensor_wide WHERE device_id = 1 AND time > NOW() - INTERVAL '1 hour' ORDER BY time DESC LIMIT 100;"

echo "=== Aggregation: avg per minute last 6 hours ==="
$PG "EXPLAIN (ANALYZE, BUFFERS) SELECT time_bucket('1 minute', time) AS minute, avg(value) FROM sensor_readings WHERE metric='cpu' AND time > NOW() - INTERVAL '6 hours' GROUP BY minute;"
```

```
$PG "EXPLAIN (ANALYZE, BUFFERS) SELECT time_bucket('1 minute', time) AS minute, avg(value) FROM sensor_plain WHERE metric='cpu' AND time > NOW() - INTERVAL '6 hours' GROUP BY minute;"
$PG "EXPLAIN (ANALYZE, BUFFERS) SELECT time_bucket('1 minute', time) AS minute, avg(cpu_percent) FROM sensor_wide WHERE time > NOW() - INTERVAL '6 hours' GROUP BY minute;"

echo "=== Point lookup: latest per device ==="
$PG "EXPLAIN (ANALYZE, BUFFERS) SELECT DISTINCT ON (device_id) device_id, time, value FROM sensor_readings WHERE metric='cpu' ORDER BY device_id, time DESC;"
$PG "EXPLAIN (ANALYZE, BUFFERS) SELECT DISTINCT ON (device_id) device_id, time, value FROM sensor_plain WHERE metric='cpu' ORDER BY device_id, time DESC;"
$PG "EXPLAIN (ANALYZE, BUFFERS) SELECT DISTINCT ON (device_id) device_id, time, cpu_percent FROM sensor_wide ORDER BY device_id, time DESC;"
```

Run:

```
chmod +x compare_perf.sh
./compare_perf.sh
```

Interpret the `EXPLAIN ANALYZE` outputs by comparing:

- total runtime (`Execution Time` / `Actual time`)
- number of heap blocks / buffer reads (`shared read` vs `shared hit`)
- whether sequential scans or index scans were used
- number of rows processed

Practical tips & accuracy checks

- If your dataset is small, the planner may choose different plans; to see hypertable benefits clearly, generate enough data so that multiple chunks exist (increase `rows_per_device`).
- Pick a realistic `chunk_time_interval` for production: chunks with a size of a few MBs up to a few GBs (not too small nor too big). For heavy ingestion, shorter chunk_time_interval reduces scanning.
- Use `timescaledb_information.hypertables` and `timescaledb_information.chunks` to inspect hypertable metadata:

```
SELECT * FROM timescaledb_information.hypertables;
```

SELECT table_name, range_start, range_end FROM timescaledb_information.chunks WHERE hypertable_name='sensor_readings' ORDER BY range_start LIMIT 10;

- Use `\d+ sensor_readings` in psql to inspect indexes and inherited chunk details.

Cleanup (optional)

To remove test data (keep hypertable definitions):

docker exec -i timescaledb psql -U admin -d metricsdb -c "TRUNCATE sensor_readings, sensor_wide, sensor_plain;"

To drop test tables completely:

docker exec -i timescaledb psql -U admin -d metricsdb -c "DROP TABLE IF EXISTS sensor_readings, sensor_wide, sensor_plain;"

Summary / Learning outcomes

By completing this module you will:

- Understand narrow vs wide schema tradeoffs for time-series data.
- Know how TimescaleDB partitions data by time (hypertables) and optionally by space.
- Be able to create hypertables, insert large synthetic datasets and run practical queries.
- Measure and compare query performance (hypertable vs plain Postgres table) using `EXPLAIN (ANALYZE, BUFFERS)`.

This Module 2 builds directly on the Module 1 setup (Postgres + TimescaleDB running in Docker). If you followed Module 1's instructions to bring up the stack, you are ready to run everything above.