Module 3 - Ingesting Data

Prerequisites (same stack as Module 1)
This module assumes you already have the Docker Compose stack (TimescaleDB / InfluxDB / Grafana) up from Module 1 and are using the same credentials (admin / admin123, DB `metricsdb`). If you followed Module 1, you're ready.

Files/scripts we will use in this module (save in the same folder as your `docker-compose.yml` unless noted):

- `generate_data.py` — generate CSV (`cpu_data.csv`) and Influx line-protocol (`cpu.lp`) for 1,000,000 rows.
- `stream_insert.py` — Python batched streaming insert (psycopg2).
- `insert_stream_node.js` — Node.js batched streaming insert (`pg`).
- `run_benchmarks.sh` — orchestrator: runs COPY into plain Postgres & TimescaleDB, writes to Influx, and runs a short streaming test.
- `cleanup_benchmarks.sh` — drop tables and remove generated files.

A. Key concepts (brief)

- **Batch inserts**: `COPY` or multi-row `INSERT` — fastest for bulk historical loads. Use when you have files or periodic batches.
- **Streaming inserts**: one-by-one or batched single-statement streaming (clients like Python/Node.js, Kafka connectors). Use for real-time ingestion.
- **Best practice**: For very large loads use `COPY` into a plain table or hypertable, disable or delay heavy indexes during load, then recreate/analyze indexes. For real-time, send data in batches (commits of e.g., 1k–10k rows) to reduce transaction overhead.

For background on hypertables and schema choices, see Module 2.

B. SQL: create target tables & hypertables

Run these once (idempotent) to prepare targets:

```
docker exec -i timescaledb psql -U admin -d metricsdb -v ON_ERROR_STOP=1 <<'SQL'
-- plain Postgres table for bulk copy comparison
CREATE TABLE IF NOT EXISTS sensor_plain_ingest (
  time TIMESTAMPTZ NOT NULL,
```

```
  device_id INT,
  value DOUBLE PRECISION
);

-- TimescaleDB hypertable target
CREATE TABLE IF NOT EXISTS sensor_ingest (
  time TIMESTAMPTZ NOT NULL,
  device_id INT,
  value DOUBLE PRECISION
);
SELECT create_hypertable('sensor_ingest', 'time', if_not_exists => TRUE);

-- Streaming target (narrow-style)
CREATE TABLE IF NOT EXISTS sensor_stream (
  time TIMESTAMPTZ NOT NULL,
  device_id INT,
  metric TEXT,
  value DOUBLE PRECISION
);
SELECT create_hypertable('sensor_stream', 'time', if_not_exists => TRUE);

-- Helpful index for query testing (create if you want to run queries later)
CREATE INDEX IF NOT EXISTS idx_sensor_ingest_device_time ON sensor_ingest (device_id,
time DESC);
CREATE INDEX IF NOT EXISTS idx_sensor_stream_device_metric_time ON sensor_stream
(device_id, metric, time DESC);
SQL
```

(These commands assume the `timescaledb` container and credentials from Module 1.)

C. File: `generate_data.py` (creates CSV + Influx LP)

Save as `generate_data.py`:

```
#!/usr/bin/env python3
# generate_data.py
# Generates cpu_data.csv (time,device_id,value) and cpu.lp (Influx line protocol)
from datetime import datetime, timezone, timedelta
import random

ROW_COUNT = 1_000_000
CSV_PATH = "cpu_data.csv"
LP_PATH = "cpu.lp"
```

```
start = datetime.utcnow().replace(tzinfo=timezone.utc)

def iso_ts(dt):
    return dt.isoformat()

def ns_epoch(dt):
    return str(int(dt.timestamp() * 1_000_000_000))

def main():
    print(f"Generating {ROW_COUNT:,} rows...")
    with open(CSV_PATH, "w", buffering=1_000_000) as csvf, open(LP_PATH, "w",
buffering=1_000_000) as lpf:
        for i in range(ROW_COUNT):
            ts = start + timedelta(seconds=i)
            device_id = random.randint(1, 10)
            value = round(random.uniform(10.0, 90.0), 2)
            csvf.write(f"{iso_ts(ts)},{device_id},{value}\n")
            lpf.write(f"cpu,device_id={device_id} value={value} {ns_epoch(ts)}\n")
    print("Done: cpu_data.csv and cpu.lp created.")

if __name__ == '__main__':
    main()
```

Run:

```
chmod +x generate_data.py
./generate_data.py
```

Note: generating 1,000,000 rows takes disk space and some time; running on a laptop may take a few minutes. The CSV is ready for `COPY` and the LP file for Influx `influx write`.

D. Fast bulk load: use `COPY` (CSV)

Copy into container and run `\COPY` inside the container (reliable)

```
# copy CSV into timescaledb container filesystem
docker cp cpu_data.csv timescaledb:/tmp/cpu_data.csv

# load into plain Postgres table
docker exec -i timescaledb psql -U admin -d metricsdb -c "\COPY sensor_plain_ingest(time,
device_id, value) FROM '/tmp/cpu_data.csv' CSV"
```

```
# load into TimescaleDB hypertable
docker exec -i timescaledb psql -U admin -d metricsdb -c "\COPY sensor_ingest(time,
device_id, value) FROM '/tmp/cpu_data.csv' CSV"
```

`COPY` is significantly faster than per-row `INSERT`. Use it for historical/one-off loads.

E. InfluxDB write (line-protocol)

You can stream the LP file to the Influx write command (ns precision):

```
# copy to container (optional) or stream from host
docker cp cpu.lp influxdb:/tmp/cpu.lp

# preferred: stream from host so you don't depend on container path
docker exec -i influxdb influx write --org example-org --bucket example-bucket --precision ns - <
cpu.lp
```

Influx writes are optimized for time-series; results will vary with container resources and Influx settings.

F. Python streaming client (batched) — `stream_insert.py`

Save as `stream_insert.py`:

```
#!/usr/bin/env python3
# stream_insert.py
# Batched streaming insert into a table (sensor_stream or sensor_ingest).
import argparse
import time
import random
import psycopg2
from datetime import datetime, timedelta, timezone

def parse_args():
    p = argparse.ArgumentParser()
    p.add_argument("--dsn", default="dbname=metricsdb user=admin password=admin123
host=localhost port=5432")
    p.add_argument("--table", default="sensor_stream")
    p.add_argument("--rows", type=int, default=1000000)
    p.add_argument("--batch", type=int, default=1000)
    return p.parse_args()
```

```
def main():
    args = parse_args()
    conn = psycopg2.connect(args.dsn)
    cur = conn.cursor()
    start_time = datetime.utcnow().replace(tzinfo=timezone.utc)
    t0 = time.time()
    buffer = []
    inserted = 0
    for i in range(args.rows):
        ts = start_time + timedelta(seconds=i)
        device_id = random.randint(1, 10)
        value = round(random.uniform(10.0, 90.0), 2)
        if args.table == "sensor_stream":
            buffer.append((ts, device_id, 'cpu', value))
        else:
            buffer.append((ts, device_id, value))
        if len(buffer) >= args.batch:
            if args.table == "sensor_stream":
                vals = ",".join(cur.mogrify("(%s,%s,%s,%s)", r).decode() for r in buffer)
                cur.execute("INSERT INTO sensor_stream(time, device_id, metric, value) VALUES "
+ vals)
            else:
                vals = ",".join(cur.mogrify("(%s,%s,%s)", r).decode() for r in buffer)
                cur.execute("INSERT INTO sensor_ingest(time, device_id, value) VALUES " + vals)
            conn.commit()
            inserted += len(buffer)
            buffer.clear()
    if buffer:
        if args.table == "sensor_stream":
            vals = ",".join(cur.mogrify("(%s,%s,%s,%s)", r).decode() for r in buffer)
            cur.execute("INSERT INTO sensor_stream(time, device_id, metric, value) VALUES " +
vals)
        else:
            vals = ",".join(cur.mogrify("(%s,%s,%s)", r).decode() for r in buffer)
            cur.execute("INSERT INTO sensor_ingest(time, device_id, value) VALUES " + vals)
        conn.commit()
        inserted += len(buffer)
    t1 = time.time()
    cur.close()
    conn.close()
    elapsed = t1 - t0
    print(f"Inserted {inserted} rows into {args.table} in {elapsed:.2f} s ({inserted/elapsed:.2f}
rows/s)")
```

```
if __name__ == "__main__":
    main()
```

Install dependency and run:

```
python3 -m venv venv
source venv/bin/activate
pip install psycopg2-binary

# example: run streaming test into Timescale hypertable (100k rows as a short run)
python3 stream_insert.py --table sensor_ingest --rows 100000 --batch 1000
```

Notes: default DSN expects `localhost:5432` mapping from Module 1. If Postgres is not exposed to host, run this script inside Docker network (instructions below).

G. Batched Node.js client — `insert_stream_node.js`

Save as `insert_stream_node.js` (improved, batch-insert):

```
// insert_stream_node.js
// Usage: node insert_stream_node.js [rows] [batchSize]
// Defaults: 10000 rows, 1000 batch
const { Client } = require('pg');
const ROWS = parseInt(process.argv[2] || '10000', 10);
const BATCH = parseInt(process.argv[3] || '1000', 10);
const client = new Client({
  user: process.env.PGUSER || 'admin',
  host: process.env.PGHOST || 'localhost',
  database: process.env.PGDATABASE || 'metricsdb',
  password: process.env.PGPASSWORD || 'admin123',
  port: parseInt(process.env.PGPORT || '5432', 10),
});

function randomValue() { return (20 + Math.random() * 60).toFixed(2); }
function isoNowPlusSeconds(start, i) { return new Date(start.getTime() + i * 1000).toISOString();
}

async function run() {
  await client.connect();
  console.log(`Connected. Inserting ${ROWS} rows in batches of ${BATCH}...`);
  const start = new Date();
  let batchValues = [];
```

```
  let total = 0;
  for (let i = 0; i < ROWS; i++) {
    const ts = isoNowPlusSeconds(start, i);
    const device_id = Math.floor(Math.random() * 5) + 1;
    const value = randomValue();
    batchValues.push(ts, device_id, 'cpu', value);
    if (batchValues.length >= BATCH * 4) {
      const rows = BATCH;
      const placeholders = [];
      for (let r = 0; r < rows; r++) {
        const base = r * 4;
        placeholders.push(`($${base+1},$${base+2},$${base+3},$${base+4})`);
      }
      const text = `INSERT INTO sensor_stream(time, device_id, metric, value) VALUES
${placeholders.join(',')}`;
      await client.query(text, batchValues);
      total += rows;
      batchValues = [];
    }
  }
  if (batchValues.length) {
    const rows = batchValues.length / 4;
    const placeholders = [];
    for (let r = 0; r < rows; r++) {
      const base = r * 4;
      placeholders.push(`($${base+1},$${base+2},$${base+3},$${base+4})`);
    }
    const text = `INSERT INTO sensor_stream(time, device_id, metric, value) VALUES
${placeholders.join(',')}`;
    await client.query(text, batchValues);
    total += rows;
  }
  await client.end();
  console.log(`Done: inserted ${total} rows.`);
}

run().catch(err => { console.error(err); process.exit(1); });
```

Install & run:

```
npm init -y
npm install pg
node insert_stream_node.js 50000 5000   # example
```

Where to store: put `insert_stream_node.js` in the same project folder (with `docker-compose.yml`) for convenience. If Postgres is not accessible on `localhost:5432`, run the script inside the Docker network (see next section).

H. Running scripts inside the Docker network

1) Find the TimescaleDB container name (use this exact command)

This finds a running container whose image name contains `timescale` (fallback to `timescaledb` if none found).

```
TS=$(docker ps --format '{{.Names}} {{.Image}}' | awk '/timescale/ {print $1; exit}')
if [ -z "$TS" ]; then
  # fallback — change this if your container has a different name
  TS="timescaledb"
fi
echo "Using TimescaleDB container name: $TS"
```

If this prints a name other than `timescaledb`, use that name in the following steps.

2) Get the Docker network that container sits on

This extracts the first network attached to the container and saves it in `NETWORK`:

```
NETWORK=$(docker inspect -f '{{range $k,$v := .NetworkSettings.Networks}}{{$k}} {{end}}'
"$TS" | awk '{print $1}')
if [ -z "$NETWORK" ]; then
  echo "ERROR: Could not detect network for container $TS. Run: docker inspect $TS"
  exit 1
fi
echo "Using Docker network: $NETWORK"
```

3) (Optional) Quick connectivity test from a throwaway container

This verifies that a container on the same network can reach Postgres on the TimescaleDB container:

```
docker run --rm --network "$NETWORK" -e PGPASSWORD=admin123 postgres:16 \
  psql -h "$TS" -U admin -d metricsdb -c "SELECT 'ok' as test, now();"
```

If this prints a row with `ok`, network connectivity is correct. If it fails, stop here and paste the error — we'll debug the container name / network.

4) Run the Node.js script on the same network (use the detected container name)

This runs `insert_stream_node.js` inside a temporary Node container and tells it to connect to the DB host named by the TimescaleDB container.

```
docker run --rm \
  -v "$PWD":/app -w /app \
  --network "$NETWORK" \
  -e PGHOST="$TS" -e PGUSER=admin -e PGPASSWORD=admin123 -e
PGDATABASE=metricsdb \
  node:20-bullseye \
  bash -lc "npm init -y >/dev/null && npm install pg >/dev/null && node insert_stream_node.js
10000 1000"
```

Notes:

- If `TS` is `timescaledb` this becomes identical to earlier examples; if your container has a different name the command still works because `PGHOST` is set from `TS`.
- `10000 1000` = `rows batchSize`. Change if you want.

5) Run the Python script on the same network — pass an explicit DSN (this fixes the `localhost` problem)

Important: the Python `stream_insert.py` defaults to a DSN that uses `host=localhost`. When running inside Docker you must tell it to use the TimescaleDB container host. Pass `--dsn` explicitly as below (replace `$TS` if you changed it):

```
docker run --rm \
  -v "$PWD":/app -w /app \
  --network "$NETWORK" \
  -e PGUSER=admin -e PGPASSWORD=admin123 -e PGDATABASE=metricsdb \
  python:3.11-slim \
  bash -lc "pip install --no-cache-dir psycopg2-binary >/dev/null && python3 stream_insert.py --
table sensor_ingest --rows 10000 --batch 1000 --dsn \"dbname=metricsdb user=admin
password=admin123 host=${TS} port=5432\""
```

Key point: `--dsn " ... host=${TS} ..."` forces the script to connect to the TimescaleDB container (not `localhost`). That removes the “connection refused”.

6) Verify results (from host)

After the job finishes, check counts:

```
docker exec -i "$TS" psql -U admin -d metricsdb -c "SELECT count(*) FROM sensor_ingest;"
# or if you used sensor_stream:
docker exec -i "$TS" psql -U admin -d metricsdb -c "SELECT count(*) FROM sensor_stream;"
```

I. Complete orchestrator: `run_benchmarks.sh`

Save as `run_benchmarks.sh` — this automates generation, COPY loads, Influx write, and a short streaming run:

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

PG_CONTAINER="timescaledb"
INFLUX_CONTAINER="influxdb"
PG_USER="admin"
PG_DB="metricsdb"
INFLUX_ORG="example-org"
INFLUX_BUCKET="example-bucket"
CSV_LOCAL="cpu_data.csv"
LP_LOCAL="cpu.lp"

# 1) generate data if missing
if [ ! -f "$CSV_LOCAL" ] || [ ! -f "$LP_LOCAL" ]; then
  echo "Generating data..."
  python3 generate_data.py
else
  echo "Data present, skipping generation."
fi

# copy CSV into Postgres container and LP for Influx
docker cp "$CSV_LOCAL" ${PG_CONTAINER}:/tmp/cpu_data.csv
docker cp "$LP_LOCAL" ${INFLUX_CONTAINER}:/tmp/cpu.lp

measure() {
  label="$1"; shift
  echo "---- $label ----"
  start=$(date +%s.%N)
  "$@"
  end=$(date +%s.%N)
  elapsed=$(awk "BEGIN {print ($end - $start)}")
  printf "%s elapsed: %0.3f s\n\n" "$label" "$elapsed"
```

```
}

# COPY into plain Postgres
measure "COPY -> sensor_plain_ingest" docker exec -i ${PG_CONTAINER} psql -U
${PG_USER} -d ${PG_DB} -c "\COPY sensor_plain_ingest(time, device_id, value) FROM
'/tmp/cpu_data.csv' CSV"

# COPY into Timescale hypertable
measure "COPY -> sensor_ingest (hypertable)" docker exec -i ${PG_CONTAINER} psql -U
${PG_USER} -d ${PG_DB} -c "\COPY sensor_ingest(time, device_id, value) FROM
'/tmp/cpu_data.csv' CSV"

# Influx write
measure "InfluxDB write to example-bucket" sh -c "docker exec -i ${INFLUX_CONTAINER}
influx write --org ${INFLUX_ORG} --bucket ${INFLUX_BUCKET} --precision ns - <
${LP_LOCAL}"

# Short streaming test (100k default to keep time reasonable)
echo "Running streaming insert test (100,000 rows)"
python3 stream_insert.py --table sensor_stream --rows 100000 --batch 1000

# counts
echo "Counts:"
docker exec -i ${PG_CONTAINER} psql -U ${PG_USER} -d ${PG_DB} -c "SELECT
'sensor_plain_ingest' as table, count(*) FROM sensor_plain_ingest;"
docker exec -i ${PG_CONTAINER} psql -U ${PG_USER} -d ${PG_DB} -c "SELECT
'sensor_ingest' as table, count(*) FROM sensor_ingest;"
docker exec -i ${PG_CONTAINER} psql -U ${PG_USER} -d ${PG_DB} -c "SELECT
'sensor_stream' as table, count(*) FROM sensor_stream;"

echo "Benchmark run complete."
```

Make executable & run:

```
chmod +x run_benchmarks.sh
./run_benchmarks.sh
```

Notes:

- `run_benchmarks.sh` uses `docker cp` and `\COPY` for best performance and to avoid host/volume permission issues.

- It runs a smaller streaming test (100k) so the full run finishes in reasonable time; change `--rows` to 1_000_000 if you want the full streaming test (expect it to take much longer).

J. Cleanup: `cleanup_benchmarks.sh`

Save as `cleanup_benchmarks.sh`: (I modified and improved that file after I created the video tutorial so that it will cleanup tables without any problem)

Save this file in the same project directory and run it.

```
#!/usr/bin/env bash
# cleanup_benchmarks.sh (hypertable-safe)
# Drops benchmark tables one-by-one (avoids "cannot drop a hypertable along with other objects"),
# removes files copied into containers, and optionally cleans Node artifacts.
set -euo pipefail

PG_CONTAINER=${PG_CONTAINER:-timescaledb}
INFLUX_CONTAINER=${INFLUX_CONTAINER:-influxdb}
PG_USER=${PG_USER:-admin}
PG_DB=${PG_DB:-metricsdb}

LOCAL_FILES=("cpu_data.csv" "cpu.lp")
CONTAINER_PG_TMP="/tmp/cpu_data.csv"
CONTAINER_INFLUX_TMP="/tmp/cpu.lp"

KEEP_NODE=${KEEP_NODE:-false}  # set KEEP_NODE=true to preserve package.json etc

TABLES=("sensor_plain_ingest" "sensor_ingest" "sensor_stream")

echo "=== Cleanup start ==="
echo "Using Postgres container: ${PG_CONTAINER}, DB: ${PG_DB}"

# show hypertables for info
echo "Hypertables currently present (if any):"
docker exec -i "${PG_CONTAINER}" psql -U "${PG_USER}" -d "${PG_DB}" -c "SELECT hypertable_name FROM timescaledb_information.hypertables;" || true

echo "Dropping tables individually (safe for hypertables)..."
for t in "${TABLES[@]}"; do
  echo "Attempting: DROP TABLE IF EXISTS ${t};"
```

```
  if docker exec -i "${PG_CONTAINER}" psql -U "${PG_USER}" -d "${PG_DB}" -c "DROP
TABLE IF EXISTS ${t};" ; then
    echo "  Dropped ${t} (or it did not exist)."
  else
    echo "  DROP TABLE failed for ${t} — retrying with CASCADE ..."
    # try again with CASCADE (will remove dependent objects)
    docker exec -i "${PG_CONTAINER}" psql -U "${PG_USER}" -d "${PG_DB}" -c "DROP
TABLE IF EXISTS ${t} CASCADE;"
    echo "  Dropped ${t} with CASCADE."
  fi
done

echo "Removing temporary files inside containers (if present)..."
if docker exec "${PG_CONTAINER}" test -f "${CONTAINER_PG_TMP}" >/dev/null 2>&1; then
  docker exec -i "${PG_CONTAINER}" bash -lc "rm -f '${CONTAINER_PG_TMP}' && echo
'Removed ${CONTAINER_PG_TMP} from ${PG_CONTAINER}'"
else
  echo "  ${CONTAINER_PG_TMP} not found in ${PG_CONTAINER}"
fi

if docker exec "${INFLUX_CONTAINER}" test -f "${CONTAINER_INFLUX_TMP}" >/dev/null
2>&1; then
  docker exec -i "${INFLUX_CONTAINER}" bash -lc "rm -f '${CONTAINER_INFLUX_TMP}' &&
echo 'Removed ${CONTAINER_INFLUX_TMP} from ${INFLUX_CONTAINER}'"
else
  echo "  ${CONTAINER_INFLUX_TMP} not found in ${INFLUX_CONTAINER}"
fi

echo "Removing local generated files..."
for f in "${LOCAL_FILES[@]}"; do
  if [ -f "$f" ]; then
    rm -f "$f"
    echo "  removed: $f"
  else
    echo "  not found: $f"
  fi
done

if [ "${KEEP_NODE}" = "true" ] || [ "${KEEP_NODE}" = "True" ]; then
  echo "KEEP_NODE=true -> preserving package.json, package-lock.json and node_modules (if
any)."
else
  echo "Removing Node artifacts on host (if present): package.json, package-lock.json,
node_modules"
```

```
  if [ -f package.json ]; then rm -f package.json && echo "  removed: package.json"; else echo "
package.json not found"; fi
  if [ -f package-lock.json ]; then rm -f package-lock.json && echo "  removed: package-
lock.json"; else echo "  package-lock.json not found"; fi
  if [ -d node_modules ]; then rm -rf node_modules && echo "  removed: node_modules/"; else
echo "  node_modules not found"; fi
fi

echo
echo "NOTE: If you still see hypertables in timescaledb_information.hypertables, list dependent
objects:"
echo "  docker exec -i ${PG_CONTAINER} psql -U ${PG_USER} -d ${PG_DB} -c \"SELECT *
FROM timescaledb_information.hypertables;\""
echo "If there are dependent continuous aggregates or policies, drop those objects first (e.g.,
DROP MATERIALIZED VIEW <name> or remove policies)."

echo "Cleanup complete."
echo "To remove Docker containers & volumes (destructive): docker compose down -v"
echo "=== Cleanup finished ==="
```

How to run:

Basic (delete node artifacts too):

```
chmod +x cleanup_benchmarks.sh
./cleanup_benchmarks.sh
```

Preserve Node files (if you want to keep `package.json`/`node_modules`):

```
chmod +x cleanup_benchmarks.sh
KEEP_NODE=true ./cleanup_benchmarks.sh
```

K. How to measure & interpret ingestion speed

- The scripts print wall-clock time for each major step. Calculate rows/sec: `rows / elapsed_seconds`.
- **COPY** will usually show the highest rows/sec (millions/sec on good hardware).
- **Batched multi-row INSERT** (Python/Node batching) can reach tens to hundreds of thousands/sec depending on batch size and hardware. Smaller batches → lower throughput.
- **Per-row single `INSERT` (await each query)** is much slower — avoid in benchmarks.

- For meaningful comparisons: run each test multiple times, ensure no other heavy processes are running, and ensure containers have enough CPU/memory. See Module 2 for recommended test queries to compare post-ingestion read performance.

L. Tips & tuning (practical)

- For **bulk loads**: temporarily drop heavy indexes, `VACUUM` / `ANALYZE` after load, increase `shared_buffers` and `work_mem` for the container if you control Docker Compose settings.
- For **streaming**: use connection pooling and keep batch size in a sweet spot (500–10,000) depending on memory.
- For production: use logical ingestion pipelines (Kafka, Fluentd, vector) and connectors; TimescaleDB has official connectors and integrations.

M. Where to place files & run summary

1. Put all files (`generate_data.py`, `stream_insert.py`, `insert_stream_node.js`, `run_benchmarks.sh`, `cleanup_benchmarks.sh`) in the same folder as your `docker-compose.yml` used in Module 1. This simplifies `docker cp`, and network assumptions.
2. Ensure Docker is running and your compose stack is up: `docker compose up -d`.
3. Generate data: `./generate_data.py` or `python3 generate_data.py`.
4. Run orchestrator: `chmod +x run_benchmarks.sh && ./run_benchmarks.sh`.
5. When finished, clean up: `chmod +x cleanup_benchmarks.sh && ./cleanup_benchmarks.sh`.

N. References from earlier modules

- The Docker Compose stack & credentials used throughout are the same as Module 1.
- Hypertable schema choices and benchmarking queries come from Module 2 (narrow vs wide, chunk_time_interval guidance).