TSDB005 Module 5 - Data Retention & Compression

In this module, you will:

1. Learn about **retention policies** (automatic data expiration).
2. Understand **native compression** in TimescaleDB and InfluxDB.
3. Hands-on:
    - Create a retention policy to keep only the last 30 days of data.
    - Enable compression and compare storage size.
    - Benchmark query performance before vs. after compression.

Step 1: What Are Retention Policies?

Retention policy means old data is automatically deleted once it exceeds a defined period.

- In **TimescaleDB**, this is done using **add_retention_policy()** on hypertables.
- In **InfluxDB**, retention policies (RPs) are built-in and managed at the bucket level.

Example use case: IoT temperature logs where only the **last 30 days** are needed. Older data is dropped to save disk space.

Step 2: What Is Native Compression?

- **TimescaleDB compression**:
  Compresses historical chunks of hypertables using columnar storage. Queries automatically decompress when needed.
  Benefits: reduced storage (up to 90%), faster scans of historical data.
- **InfluxDB compression**:
  InfluxDB stores time-series data using **TSM (Time-Structured Merge Tree)** files with built-in compression (no manual enable required).
  But you can manage retention + compaction policies to reduce disk usage.

Step 3: Apply Retention Policy in TimescaleDB

Connect to TimescaleDB:

```
docker exec -it timescaledb psql -U admin -d metricsdb
```

Inside psql, apply a retention policy (keep last 30 days of `sensor_data`):

```
-- Drop if exists to avoid conflicts
SELECT remove_retention_policy('sensor_data');

-- Keep only last 30 days of data
SELECT add_retention_policy('sensor_data', INTERVAL '30 days');
```

Verify:

```
SELECT * FROM timescaledb_information.jobs WHERE hypertable_name = 'sensor_data';
```

You will see a background job scheduled for retention.

Step 4: Apply Retention Policy in InfluxDB

InfluxDB v2 stores retention as a bucket property. The `influx` CLI requires a **bucket ID** when updating an existing bucket's settings.

List buckets and find the bucket ID

```
docker exec -it influxdb influx bucket list
```

Example output columns:

```
ID               Name            Retention    Shard group duration    Organization ID    Schema
Type
67d33fc7579e4f47 example-bucket  infinite     168h0m0s                8bda9e11c1bbaad2   implicit
```

Here, the **bucket ID** is `67d33fc7579e4f47`. (Do not confuse this with the Organization ID.)

Update retention using the bucket ID (30 days = 720h)

```
docker exec -it influxdb influx bucket update --id <ID> --retention 720h
```

Verify the change

```
docker exec -it influxdb influx bucket list
```

You should now see `Retention: 720h0m0s` for `example-bucket`.

Notes about authentication

- If the CLI inside the container is already configured (it normally is when you used `DOCKER_INFLUXDB_INIT_*`), the above commands will run without extra flags.
- If you see an authorization error, pass the token explicitly using `--token`:

```
docker exec -it influxdb influx bucket update --id 67d33fc7579e4f47 --retention 720h --token
<YOUR_TOKEN>
```

To get the initial setup token (if you didn’t set `DOCKER_INFLUXDB_INIT_TOKEN`), check the container logs for the token printed during initialization:

```
docker logs influxdb 2>&1 | grep -i token || true
```

Alternative: Create bucket with retention at creation time

If the bucket does not yet exist, you can create it with retention directly:

```
docker exec -it influxdb influx bucket create --name example-bucket --retention 720h
```

Alternative: Use HTTP API (curl)

If you prefer to use the HTTP API rather than the CLI (requires `TOKEN` and `BUCKET_ID`):

```
# 2592000 seconds = 30 days
curl -s -X PATCH http://localhost:8086/api/v2/buckets/67d33fc7579e4f47 \
  -H "Authorization: Token <TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{"retentionRules":[{"type":"expire","everySeconds":2592000}]}'
```

Step 5: Enable Compression in TimescaleDB

By default, hypertables are uncompressed. To enable:

```
docker exec -it timescaledb psql -U admin -d metricsdb
```

Inside psql,

```
-- Mark columns for compression (time is always segment_by)
ALTER TABLE sensor_data SET (timescaledb.compress, timescaledb.compress_segmentby =
'sensor_id');

-- Enable compression policy for chunks older than 7 days
SELECT add_compression_policy('sensor_data', INTERVAL '7 days');
```

Manually compress existing chunks:

```
-- Compress all chunks older than 7 days
SELECT compress_chunk(i.chunk_schema || '.' || i.chunk_name)
FROM timescaledb_information.chunks i
WHERE hypertable_name = 'sensor_data';
```

Check compressed chunks:

```
SELECT chunk_name, is_compressed
FROM timescaledb_information.chunks
WHERE hypertable_name = 'sensor_data';
```

Step 6: Compare Storage Size

Before compression:

```
SELECT pg_size_pretty(pg_total_relation_size('sensor_data'));
```

After compression:

```
-- Run again to see reduced size
SELECT pg_size_pretty(pg_total_relation_size('sensor_data'));
```

You should notice a significant reduction depending on data volume.

Step 7: Benchmark Query Performance

Run a range query **before compression** (insert fresh rows if needed):

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT AVG(temperature)
FROM sensor_data
WHERE time > NOW() - INTERVAL '30 days';
```

Step 8: Cleanup

When you finish experimenting:

```
# Stop services but keep data
docker compose down

# To reset everything (delete data volumes)
docker compose down -v
```

If you want to remove images:

```
docker rmi timescale/timescaledb:latest-pg16
docker rmi influxdb:2.7
docker rmi grafana/grafana-oss:11.2.0
```

Verify cleanup:

```
docker ps -a
docker volume ls
docker images
```

Summary

- **Retention policies** help automatically delete old data.
- **Compression** in TimescaleDB reduces storage and speeds up queries on historical data.
- InfluxDB applies retention at the bucket level and has built-in compression.
- You tested both retention and compression hands-on, compared storage size, and benchmarked query performance.