Module 7: Scaling TimescaleDB

In this module, you will:

1. **Understand when you need scaling**
2. **Set up a multi-node simulation** using Docker
3. **Run a simple performance test**
4. **Learn when to use scaling** in real projects
5. **Clean up**

What is Scaling and When Do You Need It?

Simple Explanation

Single-node TimescaleDB = One computer handling all your data **Distributed TimescaleDB** = Multiple computers sharing the work

Quick Decision Guide

✅ **You DON'T need scaling if:**

- Your data is under 500GB
- You get fewer than 50,000 inserts per second
- Your current setup works fine

🚀 **You DO need scaling if:**

- Your data is over 1TB
- You need more than 100,000 inserts per second
- Your queries are getting slow due to data volume

Step 1: Quick Setup

Create Project Directory

```
mkdir timescale-scaling-test
cd timescale-scaling-test
```

Create docker-compose.yml

```
services:
  # Main coordinator node
  coordinator:
    image: timescale/timescaledb:latest-pg16
    container_name: timescale-coordinator
    environment:
      - POSTGRES_USER=admin
      - POSTGRES_PASSWORD=admin123
      - POSTGRES_DB=testdb
    ports:
      - "5432:5432"
    volumes:
      - coordinator_data:/var/lib/postgresql/data

  # Worker node 1
  worker1:
    image: timescale/timescaledb:latest-pg16
    container_name: timescale-worker1
    environment:
      - POSTGRES_USER=admin
      - POSTGRES_PASSWORD=admin123
      - POSTGRES_DB=worker1db
    ports:
      - "5433:5432"
    volumes:
      - worker1_data:/var/lib/postgresql/data

  # Worker node 2
  worker2:
    image: timescale/timescaledb:latest-pg16
    container_name: timescale-worker2
    environment:
      - POSTGRES_USER=admin
      - POSTGRES_PASSWORD=admin123
      - POSTGRES_DB=worker2db
    ports:
      - "5434:5432"
    volumes:
      - worker2_data:/var/lib/postgresql/data

volumes:
  coordinator_data:
  worker1_data:
```

```
  worker2_data:
```

Start the Cluster

```
# Start all containers
docker compose up -d

# Wait 30 seconds for startup
sleep 30

# Verify all are running
docker compose ps
```

Step 2: Set Up Tables

Create Single-Node Table (Coordinator)

```
docker exec -it timescale-coordinator psql -U admin -d testdb

-- Enable TimescaleDB
CREATE EXTENSION IF NOT EXISTS timescaledb;

-- Create single-node table
CREATE TABLE sensor_data_single (
    time TIMESTAMPTZ NOT NULL,
    sensor_id INTEGER NOT NULL,
    temperature DOUBLE PRECISION,
    location TEXT
);

-- Make it a hypertable
SELECT create_hypertable('sensor_data_single', 'time');

-- Exit
\q
```

Create Distributed Simulation Tables

```
# Worker 1 (sensors 1-500)
docker exec -it timescale-worker1 psql -U admin -d worker1db -c "
CREATE EXTENSION IF NOT EXISTS timescaledb;
```

```
CREATE TABLE sensor_data_part1 (
    time TIMESTAMPTZ NOT NULL,
    sensor_id INTEGER NOT NULL CHECK (sensor_id BETWEEN 1 AND 500),
    temperature DOUBLE PRECISION,
    location TEXT
);
SELECT create_hypertable('sensor_data_part1', 'time');
"

# Worker 2 (sensors 501-1000)
docker exec -it timescale-worker2 psql -U admin -d worker2db -c "
CREATE EXTENSION IF NOT EXISTS timescaledb;
CREATE TABLE sensor_data_part2 (
    time TIMESTAMPTZ NOT NULL,
    sensor_id INTEGER NOT NULL CHECK (sensor_id BETWEEN 501 AND 1000),
    temperature DOUBLE PRECISION,
    location TEXT
);
SELECT create_hypertable('sensor_data_part2', 'time');
"
```

Step 3: Performance Test

Create Simple Test Script

Create `test_performance.py`:

```
#!/usr/bin/env python3
import psycopg2
import time
import random
from datetime import datetime, timedelta
from concurrent.futures import ThreadPoolExecutor

class ScalingTest:
    def __init__(self):
        self.locations = ['Office', 'Warehouse', 'Factory']

    def connect_to_db(self, port, database):
        return psycopg2.connect(
            host='localhost', port=port, database=database,
```

```
        user='admin', password='admin123'
    )

def generate_data(self, sensor_range, num_records=5000):
    """Generate test data for specific sensor range."""
    data = []
    start_sensor, end_sensor = sensor_range

    for _ in range(num_records):
        timestamp = datetime.now() - timedelta(minutes=random.randint(0, 1440))
        sensor_id = random.randint(start_sensor, end_sensor)
        temperature = 20 + random.uniform(-5, 15)
        location = random.choice(self.locations)
        data.append((timestamp, sensor_id, temperature, location))

    return data

def insert_to_single_node(self):
    """Insert all data to single node."""
    print("📝 Testing single-node insertion...")

    # Generate data for all sensors
    data = self.generate_data((1, 1000), 10000)

    conn = self.connect_to_db(5432, 'testdb')
    start_time = time.time()

    cursor = conn.cursor()
    cursor.executemany("""
        INSERT INTO sensor_data_single (time, sensor_id, temperature, location)
        VALUES (%s, %s, %s, %s)
    """, data)
    conn.commit()

    end_time = time.time()
    duration = end_time - start_time
    rate = len(data) / duration

    print(f"  ☑ Single-node: {len(data):,} records in {duration:.2f}s ({rate:,.0f} rec/sec)")
    conn.close()
    return rate

def insert_to_worker(self, port, database, table, sensor_range):
    """Insert data to a specific worker."""
```

```
        data = self.generate_data(sensor_range, 5000)

        conn = self.connect_to_db(port, database)
        start_time = time.time()

        cursor = conn.cursor()
        cursor.executemany(f"""
            INSERT INTO {table} (time, sensor_id, temperature, location)
            VALUES (%s, %s, %s, %s)
        """, data)
        conn.commit()

        end_time = time.time()
        duration = end_time - start_time
        rate = len(data) / duration

        conn.close()
        return {'records': len(data), 'duration': duration, 'rate': rate}

    def insert_to_distributed(self):
        """Insert data to distributed setup using parallel workers."""
        print("📝 Testing distributed insertion...")

        start_time = time.time()

        # Insert to both workers in parallel
        with ThreadPoolExecutor(max_workers=2) as executor:
            future1 = executor.submit(
                self.insert_to_worker, 5433, 'worker1db', 'sensor_data_part1', (1, 500)
            )
            future2 = executor.submit(
                self.insert_to_worker, 5434, 'worker2db', 'sensor_data_part2', (501, 1000)
            )

            result1 = future1.result()
            result2 = future2.result()

        end_time = time.time()
        total_duration = end_time - start_time
        total_records = result1['records'] + result2['records']
        total_rate = total_records / total_duration

        print(f"  ☑ Distributed: {total_records:,} records in {total_duration:.2f}s ({total_rate:,.0f}
rec/sec)")
```

```
        return total_rate

    def run_comparison(self):
        """Run the complete comparison test."""
        print("🚀 TimescaleDB Scaling Performance Test")
        print("=" * 50)

        # Test single-node
        single_rate = self.insert_to_single_node()

        # Test distributed
        distributed_rate = self.insert_to_distributed()

        # Compare results
        print(f"\n📊 Results Summary:")
        print(f"  Single-node:  {single_rate:,.0f} records/second")
        print(f"  Distributed:  {distributed_rate:,.0f} records/second")

        if distributed_rate > single_rate:
            improvement = (distributed_rate / single_rate - 1) * 100
            print(f"  🎉 Distributed is {improvement:.1f}% faster")
        else:
            difference = (single_rate / distributed_rate - 1) * 100
            print(f"  📈 Single-node is {difference:.1f}% faster")

        print(f"\n💡 Key Insights:")
        print(f"  - Distributed benefits increase with more data and workers")
        print(f"  - Parallel processing can significantly improve write throughput")
        print(f"  - Real distributed setups also provide fault tolerance")

if __name__ == "__main__":
    test = ScalingTest()
    test.run_comparison()
```

Install Requirements and Run Test

```
# Install Python PostgreSQL adapter
python3 -m venv venv
source venv/bin/activate
pip install psycopg2-binary

# Run the performance test
python test_performance.py
```

This test will show you the performance difference between single-node and parallel distributed insertion.

Step 4: Quick Status Check

Create Simple Status Checker

Create `check_status.py`:

```
#!/usr/bin/env python3
import psycopg2

def check_node(name, port, database, table=None):
    try:
        conn = psycopg2.connect(
            host='localhost', port=port, database=database,
            user='admin', password='admin123'
        )
        cursor = conn.cursor()

        if table:
            cursor.execute(f"SELECT COUNT(*) FROM {table}")
            count = cursor.fetchone()[0]
            print(f"  {name}: {count:,} records")
        else:
            print(f"  {name}: Connected")

        conn.close()
    except Exception as e:
        print(f"  {name}: Error - {e}")

print(" Cluster Status Check")
print("=" * 30)

check_node("Coordinator", 5432, "testdb", "sensor_data_single")
check_node("Worker 1", 5433, "worker1db", "sensor_data_part1")
check_node("Worker 2", 5434, "worker2db", "sensor_data_part2")

# Run status check
```

```
python check_status.py
```

Step 5: When to Use Scaling in Real Projects

Simple Decision Matrix

After seeing your test results, use this guide:

🟢 **Stick with Single-Node if:**

- Your test showed single-node was fast enough
- Your data is under 100GB
- You want operational simplicity
- Cost is a major concern

🟡 **Consider Distributed if:**

- Your test showed significant distributed benefits
- Your data is 100GB-1TB and growing
- You need high availability
- You have multiple concurrent writers

🔴 **Definitely Use Distributed if:**

- Your data exceeds 1TB
- Single-node performance is insufficient
- You need geographic distribution
- Fault tolerance is critical

Real-World Guidelines

Distributed TimescaleDB is best for:

- IoT applications with millions of sensors
- Financial trading systems with high-frequency data
- Monitoring systems across multiple data centers
- Applications requiring 99.99% uptime

Single-Node TimescaleDB is perfect for:

- Most business applications
- Development and testing environments
- Applications with moderate data volumes

- Teams wanting operational simplicity

Step 6: Quick Cleanup

Remove Everything

```
# Deactivate virtual environment
deactivate

# Stop and remove all containers and volumes
docker compose down -v

# Verify cleanup
docker compose ps

# Remove project directory
cd ..
rm -rf timescale-scaling-test

# Optional: Remove Docker images
docker rmi timescale/timescaledb:latest-pg16
```

Final Verification

```
# Check no containers remain
docker ps -a | grep timescale

# Check system usage
docker system df
```

Summary: What You Learned

🎯 **Core Concepts Mastered:**

- **Scaling basics**: When and why to scale TimescaleDB
- **Performance comparison**: Single-node vs distributed insertion
- **Real-world decision making**: Choosing the right architecture

🛠️ **Hands-On Skills:**

- Set up multi-node TimescaleDB simulation
- Ran parallel performance tests
- Measured and compared scaling benefits

🚀 **Key Takeaways:**

1. **Most applications don't need distributed TimescaleDB** - single-node handles most use cases perfectly
2. **Scaling adds complexity** - only scale when you have clear evidence of need
3. **Parallel processing works** - distributing writes across multiple nodes can improve throughput
4. **Test your specific workload** - every application has different scaling characteristics

Next Steps for Your Projects:

- Start with single-node TimescaleDB
- Monitor your performance metrics
- Scale only when you hit clear bottlenecks
- Always test before making architectural changes

You now understand TimescaleDB scaling concepts and can make informed decisions for your own projects!