Shark Bake-Off: Database Benchmarking Project

2025-12-31

Contents

Shark Bake-Off: Database Benchmarking Project Plan

Overview

Compare Neo4j (graph) vs PostgreSQL (relational) for a Knowledge Base system supporting real-time military tracking with 5k-7.5k identifier lookups per 10-second cycle.

Evaluation Matrix: - Databases: Neo4j, PostgreSQL - Languages: Python (FastAPI), Go, Java (Spring Boot) - Caching: Redis, In-memory - Load: Baseline (625 qps) and Stress (1,750 qps)

Project Structure

```
shark-bake-off/
  config/                    # Benchmark & database configuration
  schema/
     neo4j/                # Cypher constraints & indexes
     postgresql/           # DDL, indexes, stored procedures
  data/
     migration/            # Neo4j export & PostgreSQL load scripts
```

```
    generators/           # Synthetic identifier generators
implementations/
    python/               # FastAPI + asyncpg/neo4j-driver
    go/                   # HTTP server + pgx/neo4j-go-driver
    java/                 # Spring Boot + JPA/Spring Data Neo4j
benchmark/
    harness/              # Custom benchmark runner
    locust/               # Load test scripts
infrastructure/
    terraform/            # AWS/GCP/Azure deployment
    docker/               # Container definitions
monitoring/               # Prometheus & Grafana
analysis/                 # Jupyter notebooks for results
```

PostgreSQL Schema Strategy

Approach: Denormalized lookup table optimized for the primary use case (identifier -> properties):

```sql
CREATE TABLE air_instance_lookup (
    id                  BIGSERIAL PRIMARY KEY,
    mode_s              VARCHAR(8),           -- Primary lookup key
    tail_number         VARCHAR(20),
    shark_name          VARCHAR(255) NOT NULL,
    platform            VARCHAR(255),
    affiliation         VARCHAR(100),
    nationality         VARCHAR(100),
    operator            VARCHAR(255),
    air_type            VARCHAR(255),
    air_model           VARCHAR(255),
    neo4j_node_id       BIGINT
);

-- Covering index for zero-heap-fetch lookups
CREATE INDEX idx_air_lookup_mode_s_covering
    ON air_instance_lookup(mode_s)
    INCLUDE (shark_name, platform, affiliation, nationality, operator)
    WHERE mode_s IS NOT NULL;
```

This flattens the 6-level Neo4j hierarchy into a single query-optimized table.

Data Migration

1. **Export from Neo4j** (`data/migration/neo4j_exporter.py`)
   - Query AirInstance nodes with hierarchy traversal

- Export to JSON with flattened hierarchy fields
- Reference: `~/ontology-doc/export_level.py` patterns

2. **Load to PostgreSQL** (`data/migration/pg_loader.py`)
   - Load denormalized lookup table
   - Validate record counts and identifier integrity

Implementation Matrix

Scenario	Database	Language	Driver/ORM	Cache
Python-Neo4j	Neo4j	Python	neo4j-driver	None/Redis/Memory
Python-PG	PostgreSQL	Python	asyncpg	None/Redis/Memory
Go-Neo4j	Neo4j	Go	neo4j-go-driver	None/Redis
Go-PG	PostgreSQL	Go	pgx	None/Redis
Java-Neo4j	Neo4j	Java	Spring Data Neo4j	None/Redis
Java-PG	PostgreSQL	Java	Spring Data JPA	None/Redis

API Endpoint (all implementations):

```
GET /api/v1/lookup/mode-s/{mode_s}
GET /api/v1/lookup/tail-number/{tail_number}
```

Benchmark Scenarios

Scenario 1: Identifier Lookup (Primary Use Case)

Simple key-value lookup: Mode_S/Tail_Number -> properties

```yaml
identifier_lookup:
  weight: 0.80              # 80% of mixed workload
  query: "SELECT * FROM air_instance_lookup WHERE mode_s = $1"
  neo4j: "MATCH (ai:AirInstance {Mode_S: $mode_s}) RETURN ai"
```

Scenario 2: Multi-hop Traversal (Analytics)

Complex graph queries comparing Neo4j's native traversal vs PostgreSQL's recursive CTEs.

Example queries: 1. **Platform Heritage** (6-hop): Get full hierarchy from AirInstance to AirType - Neo4j: `MATCH (ai)-[:Instance|Derivative*1..6]->(ancestor) RETURN ancestor` - PostgreSQL: Recursive CTE with 6-level self-join

2. **Operator Network** (3-hop): Find all aircraft sharing an operator
   - Neo4j: `MATCH (ai)-[:Used_by]->(org)<-[:Used_by]-(other) RETURN other`
   - PostgreSQL: Two JOINs through junction table
3. **Country Fleet** (2-hop): All aircraft operated by a nationality

- Neo4j: `MATCH (ai)-[:Used_by]->(org)-[:Nation]->(c {name: $country}) RETURN ai`
- PostgreSQL: JOIN with nationality filter

```yaml
analytics_queries:
  weight: 0.15              # 15% of mixed workload
  queries:
    - name: platform_heritage
      hops: 6
    - name: operator_network
      hops: 3
    - name: country_fleet
      hops: 2
```

Scenario 3: Write Operations (Knowledge Generation)

Simulate activity logging and KB updates from microservices.

```yaml
write_operations:
  weight: 0.05              # 5% of mixed workload
  operations:
    - name: log_track_association
      type: INSERT
      table: track_activity_log
    - name: update_instance_property
      type: UPDATE
      target: air_instance
```

Load Profiles

```yaml
load_profiles:
  baseline:
    target_qps: 625          # 5k-7.5k queries per 10s
    duration_seconds: 300
    concurrent_users: 50
    workload_mix:
      identifier_lookup: 0.80
      analytics: 0.15
      writes: 0.05

  stress:
    target_qps: 1750         # 15k-20k queries per 10s
    duration_seconds: 300
    concurrent_users: 200
    workload_mix:
      identifier_lookup: 0.80
      analytics: 0.15
      writes: 0.05
```

```yaml
  analytics_heavy:
    target_qps: 500
    duration_seconds: 300
    concurrent_users: 100
    workload_mix:
      identifier_lookup: 0.50
      analytics: 0.45
      writes: 0.05

metrics:
  latency_percentiles: [50, 90, 95, 99, 99.9]
  throughput_sample_interval: 1s
  collect: [cpu, memory, connections, db_stats, query_queue_depth]
```

Knowledge Generation Architecture

Decouple real-time tracking from async knowledge workflows to prevent contention:

```
                        REAL-TIME PATH (< 10ms)

  Tracking System  > KB Lookup API  > [Redis Cache]  > [Primary DB]

  Priority: LOW LATENCY
  Connection Pool: Dedicated (50 connections)


                                    (fire-and-forget)

                      ASYNC WRITE PATH

  Track Events  > [Message Queue]  > Activity Logger  > [Activity Store]
                   (Kafka/RabbitMQ)

  Priority: THROUGHPUT
  Batching: 100 events / 1 second


                                    (scheduled)

                    ANALYTICS PATH (batch)

  Analytics Service  > [Read Replica]  > Aggregations  > [Results Store]

  Priority: ISOLATION (don't impact real-time)
```

```
  Connection Pool: Separate (20 connections to replica)
```

Activity Storage Schema

```sql
-- Separate table/database for activity logging (high write volume)
CREATE TABLE track_activity_log (
    id              BIGSERIAL PRIMARY KEY,
    track_id        VARCHAR(50) NOT NULL,
    event_type      VARCHAR(50) NOT NULL,  -- 'identified', 'updated', 'lost'
    kb_object_id    BIGINT,
    mode_s          VARCHAR(8),
    timestamp       TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    properties      JSONB                  -- Flexible event payload
);

-- Partitioned by time for efficient archival
CREATE TABLE track_activity_log_partitioned (
    LIKE track_activity_log INCLUDING ALL
) PARTITION BY RANGE (timestamp);

-- Index for recent activity queries
CREATE INDEX idx_activity_track_time
    ON track_activity_log(track_id, timestamp DESC);
```

Message Queue Integration

Benchmark will include: - **Kafka** or **RabbitMQ** for async write buffering - **Producer**: Track events published from lookup API - **Consumer**: Batch writer to activity storage - **Metrics**: Queue depth, consumer lag, write throughput

---

Cloud Deployment (AWS)

- **PostgreSQL**: RDS db.r6g.large with GP3 storage + Read Replica
- **Neo4j**: EC2 with Neo4j Enterprise or AuraDB
- **Redis**: ElastiCache r6g.large (cluster mode for HA)
- **Message Queue**: Amazon MSK (Kafka) or RabbitMQ on EC2
- **App Servers**: ECS Fargate or EC2 per language
- **Load Generator**: Separate EC2 instance running Locust

Implementation Phases

Phase 1: Foundation

- ☐ Create project structure and git repo

- ☐ Set up Docker Compose for local dev (Neo4j, PostgreSQL, Redis, Kafka)
- ☐ Design PostgreSQL schemas (lookup + activity + analytics)
- ☐ Implement Neo4j -> PostgreSQL migration
- ☐ Validate migrated data
- ☐ Create multi-hop query implementations for both DBs

Phase 2: Python Implementation

- ☐ Neo4j repository (async driver) - lookups + multi-hop queries
- ☐ PostgreSQL repository (asyncpg) - lookups + recursive CTEs
- ☐ FastAPI app with cache abstraction
- ☐ Redis and in-memory cache implementations
- ☐ Kafka producer for async activity logging
- ☐ Unit/integration tests

Phase 3: Go Implementation

- ☐ Repository interfaces (lookups + analytics)
- ☐ pgx and neo4j-go-driver implementations
- ☐ HTTP server with gorilla/mux
- ☐ Redis cache integration
- ☐ Kafka producer integration

Phase 4: Java Implementation

- ☐ Spring Boot project setup
- ☐ Spring Data Neo4j repository (lookups + traversals)
- ☐ Spring Data JPA repository (lookups + CTEs)
- ☐ Redis cache with Spring Cache
- ☐ Kafka producer with Spring Kafka

Phase 5: Activity Storage & Message Queue

- ☐ Kafka/RabbitMQ setup and configuration
- ☐ Activity consumer service (batch writer)
- ☐ Activity storage implementation
- ☐ Read replica configuration
- ☐ Consumer lag monitoring

Phase 6: Benchmark Harness

- ☐ HDR Histogram-based metrics collector
- ☐ Mixed workload generator (80/15/5 split)
- ☐ Multi-hop query benchmarks
- ☐ Locust load test scripts
- ☐ Prometheus exporters (including queue metrics)
- ☐ Grafana dashboards

Phase 7: Cloud Deployment

- ☐ Terraform modules (VPC, RDS+replica, ElastiCache, MSK, ECS)
- ☐ Container image builds
- ☐ Deploy and run all benchmark scenarios

Phase 8: Analysis

- ☐ Collect results across all scenarios (lookup, analytics, mixed, write)
- ☐ Compare Neo4j vs PostgreSQL for multi-hop queries
- ☐ Analyze write path performance (queue throughput, consumer lag)
- ☐ Generate comparison visualizations
- ☐ Write recommendations report with architecture guidance

Critical Reference Files

File	Purpose
`~/ontology-doc/export_level.py`	Neo4j query patterns, driver usage
`~/ontology-doc/levels.txt`	Hierarchy level definitions
`~/ontology-doc/.env`	Database connection config

Expected Deliverables

1. **Working implementations** in Python, Go, and Java for both databases
   - Identifier lookups with caching
   - Multi-hop traversal queries (Neo4j native vs PostgreSQL recursive CTEs)
   - Async activity logging with Kafka integration
2. **Benchmark results** for all scenarios:
   - Identifier lookup latency/throughput
   - Multi-hop query performance (2-hop, 3-hop, 6-hop)
   - Mixed workload behavior (80/15/5 lookup/analytics/writes)
   - Write path throughput (queue → storage)
3. **Architecture recommendation report**:
   - Optimal database for primary use case (identifier lookup)
   - Optimal database for analytics (multi-hop queries)
   - Hybrid architecture viability (Neo4j for analytics, PostgreSQL for lookups)
   - Caching strategy recommendation
   - Message queue configuration guidance
4. **Infrastructure as Code** for reproducible cloud deployment
   - Terraform modules for AWS
   - Docker Compose for local development
   - Monitoring dashboards (Prometheus + Grafana)
5. **Knowledge generation architecture**:

- Activity storage schema and implementation
- Message queue configuration
- Read replica setup for analytics isolation