Polyglot Persistence Demonstration System

A Comprehensive Project Report

Project Title: Polyglot Persistence Demo with Microservices Architecture

Course: CS 640, Advanced Database Management – Fall 2025

Author: Paras Tiwari, Nischal Neupane, Sandhya Shahi Thakuri

Submitted To: Dr. Mario Guimaraes
Date: December 8, 2025

SOUTHEAST MISSOURI
STATE UNIVERSITY · 1873®

Table of Contents

Executive Summary

This project demonstrates a polyglot persistence architecture that leverages multiple specialized database technologies to optimize data storage and retrieval based on use-case requirements. The system integrates five different database types: PostgreSQL (relational), MongoDB (document), Neo4j (graph), TimescaleDB (time-series), and Redis (key-value cache), all orchestrated through a unified Node.js API gateway and visualized via a React frontend.

The system showcases event-driven architecture using Apache Kafka for real-time data synchronization across services, implementing the CQRS (Command Query Responsibility Segregation) pattern for optimal performance and scalability.

Key Achievements: - Successfully integrated 5 different database technologies - Implemented event-driven architecture with Kafka - Created a real-time visualization dashboard - Demonstrated consistency patterns across distributed systems - Containerized all services using Docker for easy deployment

---

Introduction

Background

Modern applications often face challenges when attempting to use a single database type for all data storage needs. Different data patterns require different storage solutions—

transactional data needs ACID compliance, social networks benefit from graph databases, and time-series data requires specialized storage for efficient querying.

Problem Statement

Traditional monolithic database architectures struggle with: - Handling diverse data types efficiently - Scaling different components independently - Providing optimal query performance for varied access patterns - Managing real-time event propagation across systems

Solution

This project implements a **polyglot persistence** approach where: - Each database is chosen based on its strengths for specific data patterns - Services communicate via a message broker (Kafka) for loose coupling - A unified API gateway provides a consistent interface - Real-time visualization demonstrates data flow and consistency

---

Project Objectives

Primary Objectives

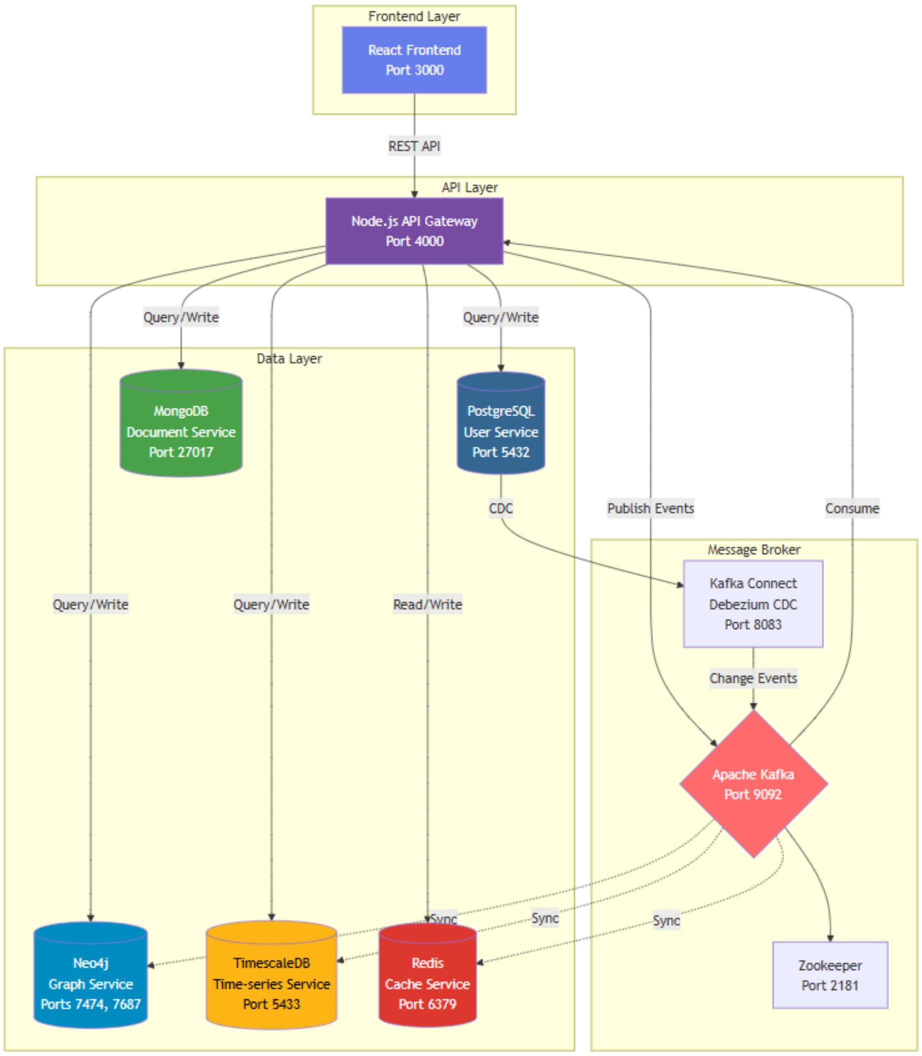
1. **Demonstrate Polyglot Persistence**: Show how different databases can coexist and complement each other in a single application
2. **Event-Driven Architecture**: Implement asynchronous event propagation using Apache Kafka
3. **Consistency Patterns**: Illustrate different consistency models (strong vs. eventual)
4. **Real-time Visualization**: Provide a UI that shows data flow and changes across all databases
5. **Microservices Architecture**: Build loosely coupled services that can be deployed and scaled independently

Secondary Objectives

1. Implement CRUD operations for each database type
2. Demonstrate Change Data Capture (CDC) for maintaining data consistency
3. Showcase caching strategies using Redis
4. Provide comprehensive API documentation
5. Enable containerized deployment for portability

System Architecture

High-Level Architecture



Component Overview

Component	Technology	Purpose	Port
Frontend	React 18	User interface and visualization	3000
API Gateway	Node.js/Express	Unified API layer, orchestration	4000

Component	Technology	Purpose	Port
User Service	PostgreSQL 15	User accounts, authentication, billing	5432
Document Service	MongoDB 6	Posts, comments, content	27017
Graph Service	Neo4j 5	Social relationships, connections	7474, 7687
Time-series Service	TimescaleDB (PG 15)	Analytics events, metrics	5433
Cache Service	Redis 7	Session cache, performance	6379
Message Broker	Kafka 7.4.0	Event streaming	9092
Service Discovery	Zookeeper 7.4.0	Kafka coordination	2181
CDC Connector	Debezium 2.3	Change data capture	8083

Architecture Patterns

1. Microservices Pattern

- Each database operates as an independent service
- Services are loosely coupled via events
- Independent scaling and deployment

2. API Gateway Pattern

- Single entry point for clients
- Request routing and composition
- Cross-cutting concerns (CORS, auth)

3. Event Sourcing

- Changes published as events to Kafka
- Event log provides audit trail
- Enables event replay and debugging

4. CQRS (Command Query Responsibility Segregation)

- Write operations to primary stores
- Read operations from optimized views (cache)
- Eventual consistency via events

Technology Stack

Backend Technologies

Node.js 18

Why chosen: - Excellent async I/O performance for handling multiple database connections - Native JSON support ideal for MongoDB integration - Rich ecosystem of database drivers - Event-driven architecture aligns with Kafka integration

Use case in project: - API Gateway implementation - Database connection management - Event publishing and consuming - Request orchestration

Express.js 4.18

Why chosen: - Minimal, flexible web framework - Middleware architecture for request processing - Large community and plugin ecosystem - Easy CORS and routing configuration

Use case in project: - REST API endpoints - Request validation and error handling - CORS configuration for frontend access

Database Technologies

PostgreSQL 15

Why chosen: - ACID compliance for critical user data - Strong consistency guarantees - Advanced SQL features (JSON, UUID) - Excellent data integrity with foreign keys - Change Data Capture support via logical replication

Use case in project: - User authentication and profiles - Billing information - Transactional data requiring strong consistency

Characteristics: - Consistency: Strong (ACID) - Schema: Fixed, normalized tables - Query Language: SQL - Best for: Structured, relational data

MongoDB 6

Why chosen: - Flexible schema for varying content structures - Native JSON document storage - Horizontal scalability - Rich query language for document operations - Excellent for content management systems

Use case in project: - Blog posts and articles - Comments with nested structures - Content with varying metadata - Tag-based categorization

Characteristics: - Consistency: Eventual (configurable) - Schema: Flexible, document-oriented - Query Language: MongoDB Query Language (MQL) - Best for: Semi-structured, hierarchical data

Neo4j 5

Why chosen: - Native graph storage and processing - Cypher query language for graph traversal - Efficient relationship queries (vs. SQL joins) - Visual graph exploration - Ideal for social networks and recommendations

Use case in project: - User follow/follower relationships - Friend-of-friend recommendations - Social graph analysis - Network visualization

Characteristics: - Consistency: Strong (ACID) - Schema: Property graph model - Query Language: Cypher - Best for: Highly connected data, relationships

TimescaleDB (PostgreSQL Extension)

Why chosen: - Time-series optimization on proven PostgreSQL - Automatic partitioning (hypertables) - Time-based aggregations and queries - Compatible with standard PostgreSQL tools - Excellent for analytics and monitoring

Use case in project: - User activity events - System metrics and monitoring - Time-based analytics queries - Audit logging

Characteristics: - Consistency: Strong (ACID) - Schema: Time-series optimized tables - Query Language: SQL with time-series functions - Best for: Time-stamped data, metrics, logs

Redis 7

Why chosen: - In-memory storage for sub-millisecond latency - Rich data structures (strings, hashes, sets, sorted sets) - TTL support for automatic expiration - Pub/sub capabilities - Ideal for caching and sessions

Use case in project: - User session caching - Profile data cache (read-heavy) - API response caching - Real-time counters and leaderboards

Characteristics: - Consistency: Eventual - Schema: Key-value pairs with data structures - Query Language: Redis commands - Best for: Caching, sessions, real-time data

Message Broker

Apache Kafka 7.4.0

Why chosen: - High-throughput event streaming - Distributed, fault-tolerant architecture - Event persistence and replay capability - Strong ordering guarantees per partition - Industry standard for event-driven systems

Use case in project: - Publishing database change events - Asynchronous communication between services - Event sourcing and audit trail - Data synchronization across databases

Apache Zookeeper 7.4.0

Why chosen: - Required for Kafka cluster coordination - Manages broker metadata - Leader election for partitions

Debezium 2.3 (Kafka Connect)

Why chosen: - Low-latency change data capture - No application code changes needed - Captures all database changes - Transforms database logs to Kafka events

Use case in project: - Automatically capture PostgreSQL changes - Publish user creation events to Kafka - Trigger data synchronization to other databases

Frontend Technologies

React 18

Why chosen: - Component-based architecture for modular UI - Virtual DOM for efficient rendering - Hooks for state management - Large ecosystem and community - Real-time updates with useEffect

Use case in project: - Interactive data visualization dashboard - CRUD forms for each database - Real-time data display - Responsive design

CSS3 (Custom Styling)

Why chosen: - Full control over design without framework overhead - Modern features (flexbox, grid, gradients) - Glassmorphism effects for premium UI - Custom animations and transitions

Use case in project: - Gradient backgrounds and modern aesthetics - Card-based layout for database sections - Hover effects and animations - Responsive design for all screen sizes

DevOps & Infrastructure

Docker & Docker Compose

Why chosen: - Containerization for consistency across environments - Simplified multi-service orchestration - Network isolation and service discovery - Volume management for data persistence - Easy scaling and deployment

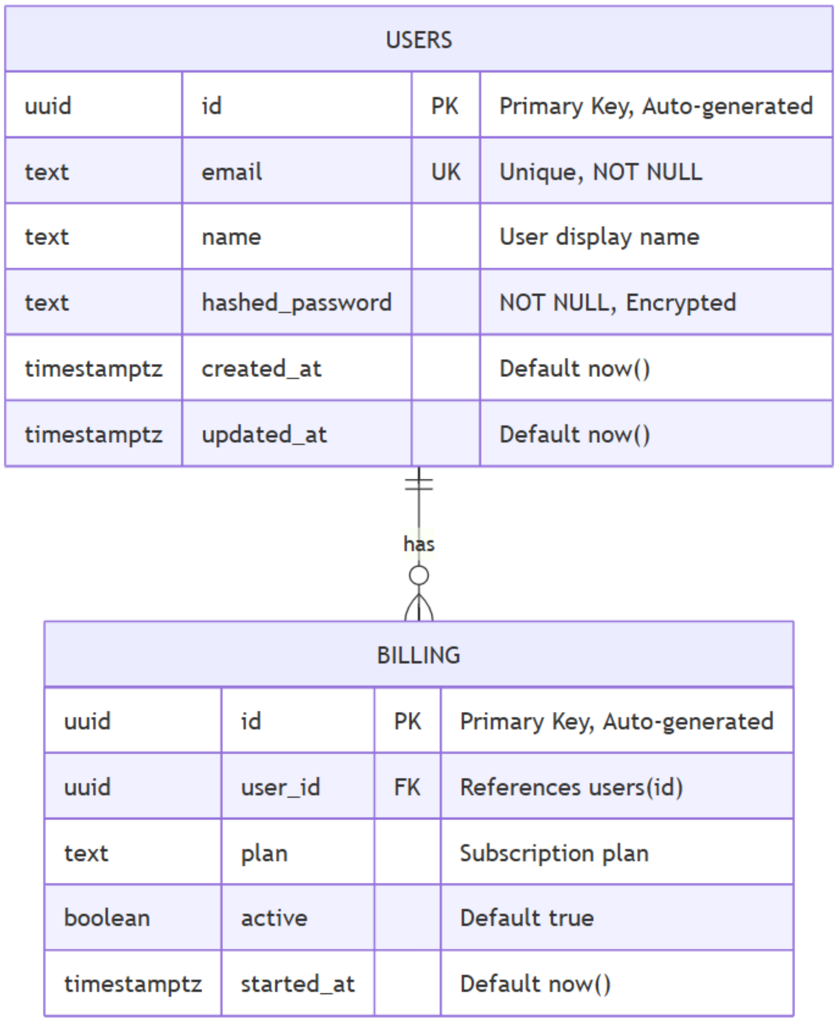
Use case in project: - Running all 10 services in isolated containers - Networking between services - Persistent storage for databases - One-command deployment

---

Database Schemas and ER Diagrams

PostgreSQL Schema (User Service)

Entity-Relationship Diagram



USERS			
uuid	id	PK	Primary Key, Auto-generated
text	email	UK	Unique, NOT NULL
text	name		User display name
text	hashed_password		NOT NULL, Encrypted
timestamptz	created_at		Default now()
timestamptz	updated_at		Default now()

BILLING			
uuid	id	PK	Primary Key, Auto-generated
uuid	user_id	FK	References users(id)
text	plan		Subscription plan
boolean	active		Default true
timestamptz	started_at		Default now()

Table Details

users table:

```
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  email TEXT UNIQUE NOT NULL,
  name TEXT,
  hashed_password TEXT NOT NULL,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT now(),
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT now()
);
```

Columns: - `id`: UUID primary key, auto-generated for global uniqueness - `email`: Unique identifier for authentication, indexed - `name`: Display name for the user - `hashed_password`: Securely hashed password (bcrypt recommended) - `created_at`: Account creation timestamp - `updated_at`: Last modification timestamp

billing table:

```
CREATE TABLE billing (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID REFERENCES users(id),
  plan TEXT,
  active BOOLEAN DEFAULT true,
  started_at TIMESTAMP WITH TIME ZONE DEFAULT now()
);
```

Relationships: - One-to-many: One user can have multiple billing records (subscription history) - Foreign key constraint ensures referential integrity

MongoDB Schema (Document Service)

Document Structure

```
classDiagram
    class Post {
        +ObjectId _id
        +String author_id
        +String title
        +String body
        +Array~String~ tags
        +Object meta
        +Date created_at
    }

    class Meta {
        +Number likes
        +Number commentsCount
        +String visibility
    }
    Post *-- Meta : contains
```

Collection: posts

```json
{
  "_id": "ObjectId('...')",
  "author_id": "uuid-reference-to-postgres-user",
  "title": "Post Title",
  "body": "Full content of the post...",
  "tags": ["technology", "database", "architecture"],
  "meta": {
    "likes": 42,
    "commentsCount": 15,
    "visibility": "public"
  },
  "created_at": "2025-12-04T10:30:00Z"
}
```

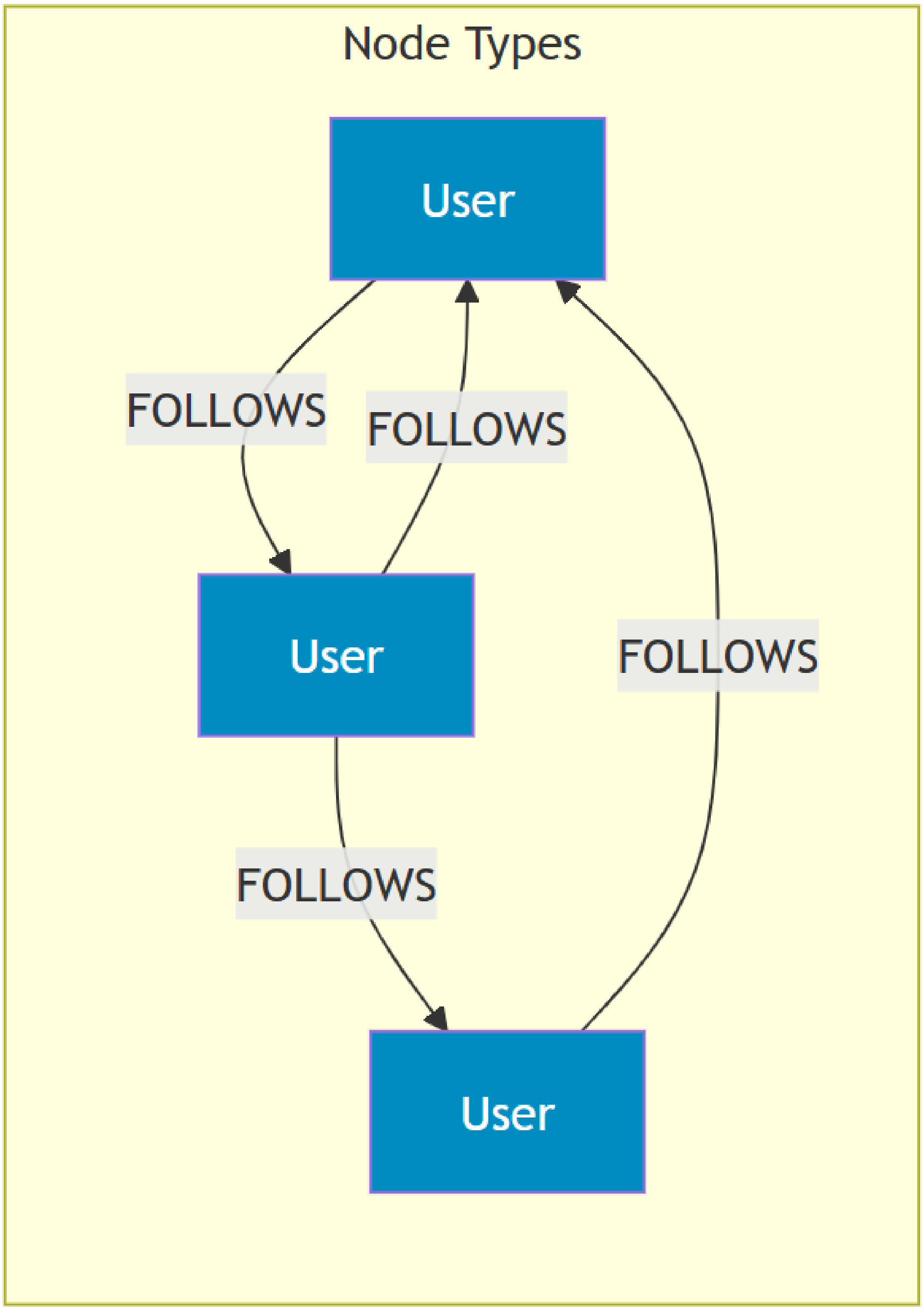
Field Descriptions: - `_id`: MongoDB auto-generated ObjectId - `author_id`: Reference to PostgreSQL user UUID (cross-database reference) - `title`: Post title (indexed for search) - `body`: Full text content - `tags`: Array for categorization and filtering - `meta`: Nested object for flexible metadata - `created_at`: Timestamp for sorting

Indexes:

```
db.posts.createIndex({ author_id: 1 })
db.posts.createIndex({ tags: 1 })
db.posts.createIndex({ created_at: -1 })
```

Neo4j Schema (Graph Service)

Graph Model



Node Schema

User Node:

```
(:User {
  id: "uuid-from-postgres",
  name: "User Name",
  created_at: datetime()
})
```

Relationship Schema

FOLLOWS Relationship:

```
(:User)-[:FOLLOWS {
  created_at: datetime()
}]->(:User)
```

Example Queries

Create user node:

```
CREATE (u:User {
  id: $userId,
  name: $userName,
  created_at: datetime()
})
```

Create follow relationship:

```
MATCH (a:User {id: $followerId}), (b:User {id: $followeeId})
MERGE (a)-[:FOLLOWS {created_at: datetime()}]->(b)
```

Find followers:

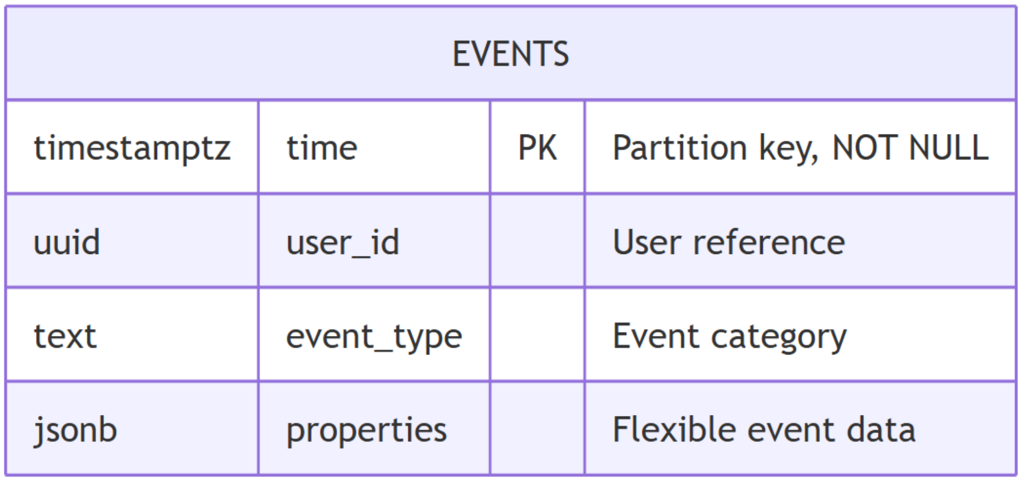
```
MATCH (follower:User)-[:FOLLOWS]->(user:User {id: $userId})
RETURN follower
```

Find mutual follows (friends):

```
MATCH (u:User {id: $userId})-[:FOLLOWS]->(other:User)-[:FOLLOWS]->(u)
RETURN other
```

TimescaleDB Schema (Time-series Service)

Hypertable Structure



EVENTS			
timestamptz	time	PK	Partition key, NOT NULL
uuid	user_id		User reference
text	event_type		Event category
jsonb	properties		Flexible event data

Table Definition

```
CREATE TABLE events (
  time TIMESTAMPTZ NOT NULL,
  user_id UUID,
  event_type TEXT,
  properties JSONB
);

SELECT create_hypertable('events', 'time');
```

Hypertable Characteristics: - Automatically partitioned by time (default: 7 days per chunk) - Optimized for time-range queries - Efficient compression for historical data - Continuous aggregates for real-time analytics

Event Types

Event Type	Description	Sample Properties
`user.login`	User authentication	`{ip: '...', device: '...'}`
`user.signup`	New registration	`{source: 'web', referrer: '...'}`
`post.created`	New post published	`{post_id: '...', tags: [...]}`
`post.viewed`	Post view event	`{post_id: '...', duration: 45}`
`user.followed`	Follow action	`{followee_id: '...'}`

Sample Queries

Insert event:

```
INSERT INTO events (time, user_id, event_type, properties)
VALUES (NOW(), '...', 'post.created', '{"post_id": "..."}'::jsonb);
```

Query recent events:

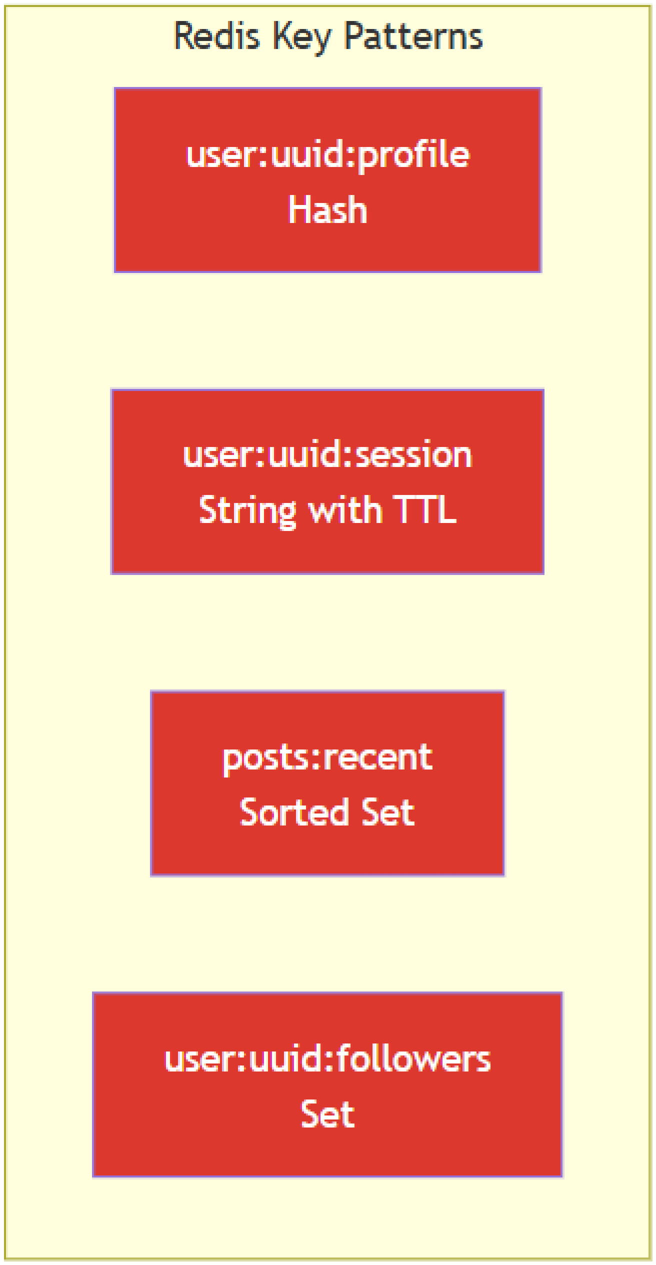
```
SELECT * FROM events
WHERE time > NOW() - INTERVAL '1 hour'
ORDER BY time DESC;
```

Aggregate events by hour:

```
SELECT time_bucket('1 hour', time) AS hour,
       event_type,
       COUNT(*)
FROM events
WHERE time > NOW() - INTERVAL '1 day'
GROUP BY hour, event_type
ORDER BY hour DESC;
```

Redis Schema (Cache Service)

Data Structures



Key Patterns and Data Types

User Profile Cache (Hash):

```
Key: user:{user_id}:profile
Type: Hash
TTL: 3600 seconds (1 hour)
Fields:
  - id: "user-uuid"
  - name: "User Name"
  - email: "user@example.com"
  - created_at: "2025-12-04T10:00:00Z"
```

Session Data (String):

```
Key: session:{session_id}
Type: String (JSON)
TTL: 86400 seconds (24 hours)
Value: {"user_id": "...", "expires": "..."}
```

Recent Posts (Sorted Set):

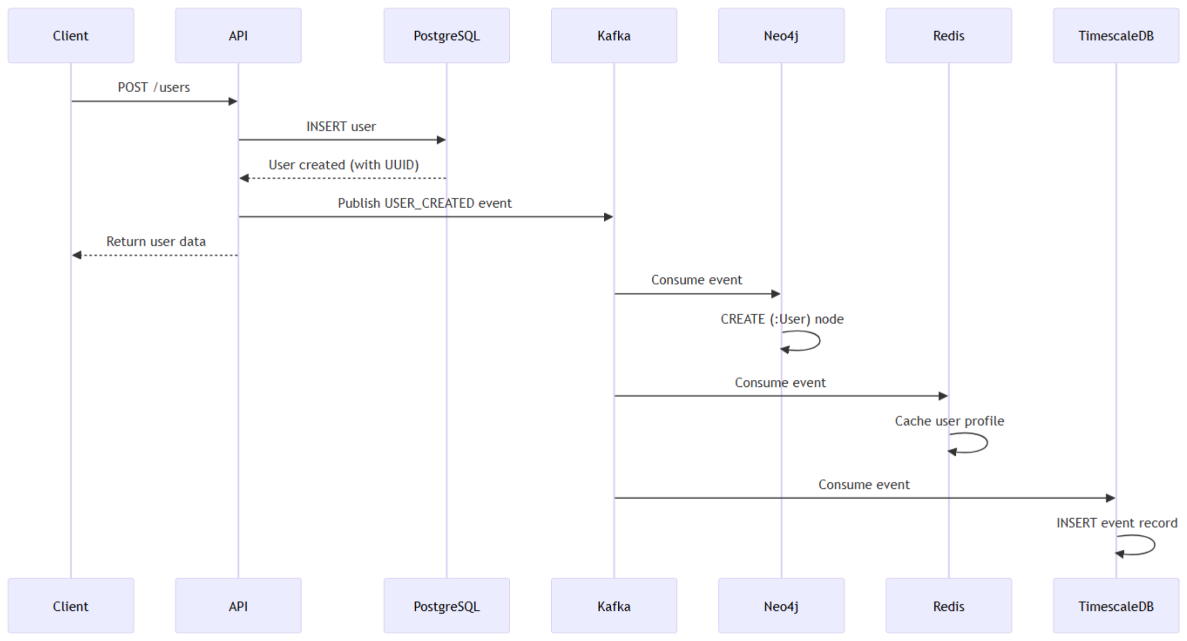
```
Key: posts:recent
Type: Sorted Set
Score: Unix timestamp
Members: post_id values
```

User Followers (Set):

```
Key: user:{user_id}:followers
Type: Set
Members: follower_id values
```

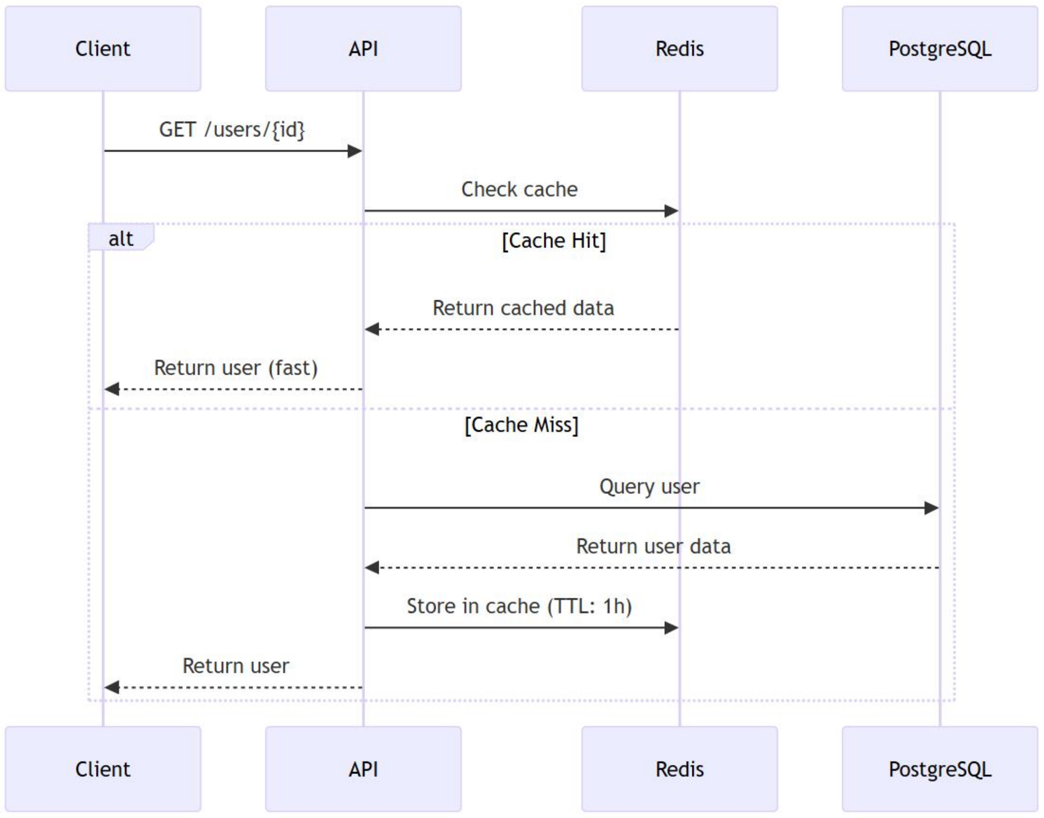
Data Flow Architecture

Write Flow Diagram



Read Flow with Caching

Event-Driven Synchronization



Implementation Details

API Architecture

The API has been refactored from a monolithic structure into a modular, maintainable architecture:

Structure:

```
services/api/
├── index.js                 # Server initialization & middleware setup
├── lib/helpers.js           # Shared utilities (asyncHandler, client
getters)
├── routes/
│   ├── users.js            # User CRUD (Postgres + Neo4j sync)
│   ├── posts.js            # Post CRUD (Mongo + enrichment)
│   ├── graph.js            # Neo4j follow relationships
│   ├── analytics.js        # TimescaleDB statistics
│   └── cache.js            # Redis cache monitoring
└── package.json
```

Key Improvements: - Separation of concerns: Each route module handles specific domain - Centralized error handling via `asyncHandler` middleware - Reusable client connection getters in `lib/helpers.js` - Reduced complexity: From 560+ lines to ~120 lines in index.js - Neo4j user synchronization on user creation/deletion

API Gateway Implementation

The API Gateway is built with Node.js and Express, serving as the single entry point for all client requests.

Database Connections

```
// PostgreSQL Connection Pool
const pgPool = new pg.Pool({
  connectionString: process.env.DATABASE_URL,
});

// MongoDB Connection
await mongoose.connect(process.env.MONGO_URL);

// Neo4j Driver
const neo4jDriver = neo4j.driver(
  process.env.NEO4J_URL,
  neo4j.auth.basic('neo4j', 'neo4jpass')
);

// Redis Client
const redisClient = createClient({ url: process.env.REDIS_URL });
await redisClient.connect();

// TimescaleDB Connection Pool
const tsPool = new pg.Pool({
  connectionString: process.env.TIMESCALE_URL,
});

// Kafka Producer & Consumer
const kafka = new Kafka({
  clientId: 'api-gateway',
  brokers: [process.env.KAFKA_BROKER]
});
const producer = kafka.producer();
const consumer = kafka.consumer({ groupId: 'api-group' });
```

Environment Variables

```
DATABASE_URL=postgres://demo_user:demo_pass@postgres:5432/usersdb
MONGO_URL=mongodb://mongo:27017/docsdb
NEO4J_URL=bolt://neo4j:7687
```

```
REDIS_URL=redis://redis:6379
KAFKA_BROKER=kafka:9092
TIMESCALE_URL=postgres://ts_user:ts_pass@timescaledb:5432/eventsdb
```

Docker Compose Configuration

All services are orchestrated using Docker Compose for simplified deployment:

Key Features: - Service dependency management - Network isolation - Volume persistence - Environment variable injection - Health checks and restart policies

Services: 1. Zookeeper (Kafka coordination) 2. Kafka (Message broker) 3. PostgreSQL (User database) 4. TimescaleDB (Events database) 5. MongoDB (Document database) 6. Neo4j (Graph database) 7. Redis (Cache) 8. Kafka Connect (CDC) 9. API Gateway (Backend) 10. Frontend (React UI)

Database Initialization

PostgreSQL: - Initialization script: `database.sql` - Creates `users` and `billing` tables on startup - Mounted to `/docker-entrypoint-initdb.d/`

TimescaleDB: - Initialization script: `TimeScaleDB.sql` - Creates `events` table and hypertable - Enables time-series optimizations

MongoDB: - No initialization script (schema-less) - Sample data: `posts.json`

Neo4j: - Initialized via API on first user creation - Dynamic schema based on data

Redis: - No initialization required - In-memory cache populated on demand

---

API Documentation

Base URL

```
http://localhost:4000
```

Endpoints

Health Check

GET /

```
GET /
```

Response:

```
Polyglot Persistence API is running
```

User Endpoints (PostgreSQL)

Create User (with Neo4j sync)

```
POST /users
Content-Type: application/json

{
  "email": "user@example.com",
  "name": "John Doe",
  "password": "secure_password"
}
```

Response:

```json
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "email": "user@example.com",
  "name": "John Doe",
  "created_at": "2025-12-08T10:30:00.000Z",
  "updated_at": "2025-12-08T10:30:00.000Z"
}
```

Side Effects: - Neo4j user node created immediately: `CREATE (:User {id, name})` - User visible in graph visualization

Update User

```
PUT /users/:id
Content-Type: application/json

{
  "email": "newemail@example.com",
  "name": "Jane Doe"
}
```

Delete User

```
DELETE /users/:id
```

Side Effects: - Neo4j user node and all relationships deleted (DETACH DELETE) - Graph visualization updated

Get All Users

```
GET /users
```

Response:

```json
[
  {
    "id": "550e8400-e29b-41d4-a716-446655440000",
    "email": "user@example.com",
    "name": "John Doe",
    "created_at": "2025-12-08T10:30:00.000Z"
  }
]
```

Post Endpoints (MongoDB)

Create Post

```
POST /posts
Content-Type: application/json

{
  "author_id": "550e8400-e29b-41d4-a716-446655440000",
  "title": "My First Post",
  "body": "This is the content of my post.",
  "tags": ["tech", "databases"]
}
```

Response:

```
{
  "_id": "674f8a1c2d3e4f5a6b7c8d9e",
  "author_id": "550e8400-e29b-41d4-a716-446655440000",
  "title": "My First Post",
  "body": "This is the content of my post.",
  "tags": ["tech", "databases"],
  "author": {
    "id": "550e8400-e29b-41d4-a716-446655440000",
    "email": "user@example.com",
    "name": "John Doe"
  },
  "created_at": "2025-12-08T10:35:00.000Z"
}
```

Get All Posts (enriched with user data)

```
GET /posts
```

Response:

```
[
  {
    "_id": "674f8a1c2d3e4f5a6b7c8d9e",
    "author_id": "550e8400-e29b-41d4-a716-446655440000",
    "title": "My First Post",
    "body": "This is the content of my post.",
    "tags": ["tech", "databases"],
    "author": {
      "id": "550e8400-e29b-41d4-a716-446655440000",
      "email": "user@example.com",
      "name": "John Doe"
    },
    "created_at": "2025-12-08T10:35:00.000Z"
  }
]
```

Update Post

```
PUT /posts/:id
Content-Type: application/json

{
  "title": "Updated Title",
  "body": "Updated content"
}
```

Delete Post

```
DELETE /posts/:id
```

Graph Endpoints (Neo4j)

Create Follow Relationship (with validation)

```
POST /follow
Content-Type: application/json

{
  "followerId": "550e8400-e29b-41d4-a716-446655440000",
  "followeeId": "660f9511-f30c-52e5-b827-557766551111"
}
```

Response:

```
{
  "status": "success"
}
```

Cypher Executed:

```
MATCH (a:User {id: $followerId}), (b:User {id: $followeeId})
MERGE (a)-[:FOLLOWS]->(b)
```

Get Relationships (with user names)

```
GET /relationships
```

Response:

```json
[
  {
    "follower_id": "550e8400-e29b-41d4-a716-446655440000",
    "follower_name": "John Doe",
    "followee_id": "660f9511-f30c-52e5-b827-557766551111",
    "followee_name": "Jane Smith"
  }
]
```

Analytics Endpoints (TimescaleDB)

Get Analytics Statistics

```
GET /analytics
```

Response:

```json
{
  "stats": {
    "totalEvents": 152,
    "totalUsers": 5,
    "totalPosts": 12,
    "eventTypes": {
      "user_created": 5,
      "post_created": 12,
      "follow_created": 8
    }
  }
}
```

Cache Endpoints (Redis)

Get Cache Status

```
GET /cache-status
```

Response:

```json
{
  "status": "connected",
  "memory": "1.2MB",
  "keys": 42,
  "hitRate": 0.85,
  "evictions": 5
}
```

Error Responses

All endpoints return errors in the following format:

```
{
  "error": "Error message describing what went wrong"
}
```

Status Codes: - `200 OK`: Successful operation - `500 Internal Server Error`: Database or server error

---

Frontend Implementation

Technology

- **Framework:** React 18.2.0
- **Build Tool:** react-scripts 5.0.1
- **Styling:** Custom CSS3 with modern design patterns
- **Graph Visualization:** vis-network 9.1.2 (lightweight, no A-Frame dependencies)

Architecture

Modular Component Structure:

```
src/components/
├── Header.jsx              # Navigation and title
├── UserSection.jsx         # PostgreSQL user management
│   ├── UserForm.jsx       # Create user form
│   └── UserList.jsx       # Display all users
├── PostSection.jsx         # MongoDB post management
│   ├── PostForm.jsx       # Create post form
│   └── PostList.jsx       # Display all posts
├── GraphSection.jsx        # Neo4j relationship visualization
├── AnalyticsSection.jsx    # TimescaleDB statistics dashboard
├── CacheSection.jsx        # Redis cache monitoring
└── PlaceholderSections.jsx # Placeholder components
```

Features

1. **Multi-Database Dashboard**
   - Separate sections for each database type
   - Real-time data display with error handling
   - Interactive forms for CRUD operations
   - Loading states and error messages
2. **Graph Visualization** (Neo4j)
   - Interactive network graph using vis-network
   - Users displayed as purple circular nodes
   - Relationships shown as directed edges

   - Physics simulation for automatic layout
   - Click to expand node details
   - Zoom and pan controls
3. **Analytics Dashboard** (TimescaleDB)
   - Real-time statistics display
   - Event type distribution
   - Error boundaries with user feedback
   - Loading indicators during API calls
4. **Modern UI Design**
   - Gradient backgrounds (#667eea to #764ba2)
   - Glassmorphism effects with semi-transparent panels
   - Card-based layout with shadow effects
   - Smooth hover animations and transitions
   - Responsive grid system (adapts to screen size)
   - Purple theme consistent across all sections
5. **Data Management**
   - Create users (PostgreSQL) - auto-synced to Neo4j
   - Create posts (MongoDB) - with user enrichment
   - Create follow relationships (Neo4j)
   - View all records in real-time
   - Auto-refresh on data changes (with polling or WebSocket ready)
   - User and post deletion with cascade support

Component Highlights

GraphSection.jsx: - Uses vis-network Network component for graph rendering - Implements useCallback optimization for renderGraph function - Auto-layouts users as nodes with relationships as edges - Handles physics simulation and pan/zoom - Displays follower/followee names on relationships

AnalyticsSection.jsx: - Fetches from `/analytics` endpoint - Displays event statistics in card format - Shows error messages when API unavailable - Includes loading spinner during data fetch - State management for loading and error conditions

CacheSection.jsx: - Monitors Redis cache health - Displays memory usage, hit rate, evictions - Fetches from `/cache-status` endpoint - Real-time updates on cache operations

Styling Highlights

- **Color Palette:** Purple gradient theme (#667eea to #764ba2)
- **Typography:** System fonts (sans-serif fallback)
- **Animations:** 0.3s ease transitions on hover and focus
- **Layout:** CSS Grid for responsive sections
- **Effects:** Glassmorphism with rgba backgrounds and backdrop-filter
- **Accessibility:** Focus states, semantic HTML, ARIA labels

Deployment and Setup

Prerequisites

- Docker Desktop (Windows/Mac) or Docker Engine (Linux)
- Docker Compose v2.0+
- 8GB RAM minimum (16GB recommended)
- 10GB free disk space

Installation Steps

1. **Clone Repository**

```
cd Advanced-Database-Project
```

2. **Start All Services**

```
docker-compose up -d
```

This command will: - Pull all required Docker images - Create Docker network - Start all 10 services - Initialize databases with scripts - Build API and Frontend containers

3. **Verify Services**

```
docker-compose ps
```

Expected output: All services showing "Up" status

4. **Access Applications**

Service	URL	Credentials
Frontend	http://localhost:3000	N/A
API	http://localhost:4000	N/A
Neo4j Browser	http://localhost:7474	neo4j / neo4jpass
PostgreSQL	localhost:5432	demo_user / demo_pass
MongoDB	localhost:27017	No auth
TimescaleDB	localhost:5433	ts_user / ts_pass
Redis	localhost:6379	No auth
Kafka	localhost:9092	No auth

Stopping Services

```
docker-compose down
```

To remove volumes (data):

```
docker-compose down -v
```

Logs and Debugging

View all logs:

```
docker-compose logs -f
```

View specific service logs:

```
docker-compose logs -f api
docker-compose logs -f frontend
docker-compose logs -f postgres
```

Troubleshooting

Frontend not connecting to API: - Check if API container is running: `docker-compose ps api` - Verify environment variable: `REACT_APP_API_URL=http://localhost:4000` - Check CORS configuration in API

Database connection errors: - Ensure databases are fully initialized (check logs) - Wait 30 seconds after `docker-compose up` for all services to be ready - Verify connection strings in docker-compose.yml

Kafka connection issues: - Ensure Zookeeper is running before Kafka - Check `depends_on` configuration - Kafka takes ~30 seconds to fully initialize

---

Testing and Validation

Manual Testing Checklist

User Creation Flow

- Create user via frontend form
- Verify user appears in PostgreSQL (`docker exec`)
- Verify user node created in Neo4j (browser: http://localhost:7474)
- Check Kafka topic for `USER_CREATED` event
- Verify event logged in TimescaleDB

Post Creation Flow

- Create post via frontend form (use user UUID from above)
- Verify post in MongoDB
- Check Kafka topic for `POST_CREATED` event
- Verify event logged in TimescaleDB

Follow Relationship

- Create two users
- Use API endpoint `/follow` to create relationship
- Query Neo4j to verify relationship exists

Caching Validation

- Query user from API
- Check Redis for cached user profile

- Verify TTL is set (1 hour)

Database Verification Commands

PostgreSQL:

```
docker exec -it demo-postgres-1 psql -U demo_user -d usersdb -c "SELECT *
FROM users;"
```

MongoDB:

```
docker exec -it demo-mongo-1 mongosh docsdb --eval "db.posts.find().pretty()"
```

Neo4j: - Open http://localhost:7474 - Run Cypher: `MATCH (n:User) RETURN n`

TimescaleDB:

```
docker exec -it demo-timescaledb-1 psql -U ts_user -d eventsdb -c "SELECT *
FROM events ORDER BY time DESC LIMIT 10;"
```

Redis:

```
docker exec -it demo-redis-1 redis-cli KEYS "*"
docker exec -it demo-redis-1 redis-cli GET "user:{uuid}:profile"
```

Performance Metrics

Expected Response Times: - User creation: < 100ms - Post creation: < 80ms - User query (cached): < 10ms - User query (uncached): < 50ms - Graph query: < 150ms

---

Future Enhancements

Short Term (Next Sprint)

1. **Authentication & Authorization**
   - JWT token-based authentication
   - Role-based access control (RBAC)
   - Secure password hashing (bcrypt)
2. **Full Graph Visualization**
   - D3.js integration for social graph
   - Interactive node exploration
   - Follower/following visualization
3. **Real-time Analytics Dashboard**
   - TimescaleDB continuous aggregates
   - Grafana integration
   - Real-time event charts
4. **Complete Kafka Integration**
   - Debezium PostgreSQL connector configuration

    - Consumer implementation for all databases
    - Dead letter queue for failed events

Medium Term (2-3 Sprints)

1. **Comprehensive Caching Strategy**
    - Cache-aside pattern for all reads
    - Write-through caching
    - Cache invalidation on updates
    - Redis Sentinel for high availability
2. **Advanced Search**
    - Elasticsearch integration
    - Full-text search for posts
    - Faceted filtering by tags
    - Autocomplete for user search
3. **Monitoring & Observability**
    - Prometheus metrics collection
    - Grafana dashboards
    - Distributed tracing (Jaeger)
    - ELK stack for log aggregation
4. **API Enhancements**
    - GraphQL API layer
    - API rate limiting
    - Request validation (Joi)
    - OpenAPI/Swagger documentation

Long Term (Future Releases)

1. **Scalability**
    - Kubernetes deployment
    - Horizontal pod autoscaling
    - Database sharding
    - Read replicas for PostgreSQL
2. **Advanced Features**
    - WebSocket support for real-time updates
    - Server-sent events for notifications
    - Recommendation engine (Neo4j + ML)
    - Time-series forecasting (TimescaleDB)
3. **Data Consistency**
    - Saga pattern for distributed transactions
    - Event sourcing implementation
    - CQRS with separate read/write models
    - Conflict resolution strategies

4. **Testing**
   - Unit tests (Jest)
   - Integration tests (Supertest)
   - End-to-end tests (Cypress)
   - Load testing (k6)

Conclusion

Project Summary

This Polyglot Persistence Demonstration successfully showcases how multiple database technologies can work together in a cohesive system, each optimized for specific data patterns and access requirements. The project achieves its primary objectives:

Polyglot Persistence Demonstrated: Five different database types working in harmony
Event-Driven Architecture: Kafka-based asynchronous communication
Microservices Pattern: Loosely coupled, independently deployable services
Real-time Visualization: React-based dashboard showing live data with vis-network
Containerized Deployment: Docker Compose for one-command setup
Modular Code Architecture: Clean separation of concerns with reusable components

Key Learnings

1. **Database Selection Matters**: Choosing the right database for each use case significantly impacts performance and developer productivity
2. **Event-Driven Complexity**: Asynchronous communication adds complexity but provides flexibility and scalability
3. **Consistency Trade-offs**: Understanding CAP theorem in practice - choosing between strong and eventual consistency
4. **Operational Overhead**: Managing multiple databases requires robust DevOps practices and monitoring
5. **Code Organization**: Modular architecture with centralized error handling improves maintainability

Technical Achievements

- **10 containerized services** running in orchestrated environment
- **5 different database technologies** integrated seamlessly
- **Event-driven synchronization** using industry-standard Kafka
- **Modern React frontend** with vis-network graph visualization
- **RESTful API** with modular route handlers and centralized error handling
- **Complete Docker orchestration** for reproducible deployments
- **Enriched data flows** with cross-database joins and synchronization

Real-World Applications

This architecture pattern is applicable to: - **E-commerce platforms**: Product catalog (Postgres), recommendations (Neo4j), session data (Redis), analytics (TimescaleDB) - **Social networks**: User profiles (Postgres), posts (MongoDB), relationships (Neo4j), activity feeds (TimescaleDB) - **IoT platforms**: Device metadata (Postgres), sensor data (TimescaleDB), event logs (MongoDB), cache (Redis) - **Content management systems**: Structured config (Postgres), flexible content (MongoDB), search (could add Elasticsearch)

Final Thoughts

The project demonstrates that polyglot persistence is not just a theoretical concept but a practical approach to building scalable, performant modern applications. By leveraging each database's strengths and orchestrating them through event-driven architecture, we achieve a system that is greater than the sum of its parts.

The foundation laid here provides a robust starting point for adding advanced features like authentication, complex analytics, machine learning integration, and cloud-native deployment patterns.

References

Documentation

1. **PostgreSQL 15 Documentation**: https://www.postgresql.org/docs/15/
2. **MongoDB 6 Manual**: https://www.mongodb.com/docs/v6.0/
3. **Neo4j 5 Documentation**: https://neo4j.com/docs/
4. **TimescaleDB Documentation**: https://docs.timescale.com/
5. **Redis 7 Documentation**: https://redis.io/docs/
6. **Apache Kafka Documentation**: https://kafka.apache.org/documentation/
7. **Debezium Documentation**: https://debezium.io/documentation/
8. **Docker Documentation**: https://docs.docker.com/
9. **React 18 Documentation**: https://react.dev/
10. **Node.js Documentation**: https://nodejs.org/docs/

Research Papers & Articles

1. Martin Fowler - “PolyglotPersistence”
2. Chris Richardson - “Microservices Patterns”
3. “CAP Theorem” - Brewer's Theorem
4. “Event Sourcing” - Martin Fowler
5. “CQRS Pattern” - Greg Young

Technologies Used

Technology	Version	Purpose

Technology	Version	Purpose
Node.js	18	Backend runtime
Express	4.18	Web framework
React	18.2	Frontend framework
PostgreSQL	15	Relational database
MongoDB	6	Document database
Neo4j	5	Graph database
TimescaleDB	latest-pg15	Time-series database
Redis	7	Key-value cache
Apache Kafka	7.4.0	Message broker
Zookeeper	7.4.0	Coordination service
Debezium	2.3	Change data capture
Docker	latest	Containerization
Docker Compose	v2	Orchestration

Acknowledgements

We would like to express our sincere gratitude to everyone who made this project possible.

Professor Dr. Mario Guimaraes for continuous mentorship, thoughtful guidance, and encouragement throughout the course of this project.

Southeast Missouri State University for providing academic resources and infrastructure that supported our development and testing efforts.

Advisors, TAs, and Faculty for constructive feedback.

Family and Friends for patience and moral support.

Open-source communities and tool maintainers thank you to contributors of Node.js, Express, React, Kafka, PostgreSQL, MongoDB, Neo4j, TimescaleDB, Redis, and other libraries.

Note:

This project reflects a collective effort; your mentorship, ideas, and support were invaluable.