Enterprise Knowledge Graph Engine

A high-performance, production-ready knowledge graph system designed for complex relationship modeling, semantic reasoning, and large-scale graph analytics. Built for enterprise AI applications requiring scalable, reliable knowledge representation.

Architecture Overview

The Knowledge Graph Engine provides a comprehensive platform for storing, querying, and analyzing complex relational data through graph structures. It combines the flexibility of NoSQL with the rigor of enterprise database systems.

Core Components

- **Graph Data Model**: Immutable nodes and edges with rich metadata and versioning
- **Schema Management**: Type-safe validation with constraint enforcement
- **High-Performance Indexing**: Multiple indexing strategies for optimal query performance
- **Advanced Query Engine**: Graph algorithms and pattern matching capabilities
- **Analytics Framework**: Centrality measures, community detection, and path analysis
- **Enterprise Features**: ACID compliance, audit trails, and performance monitoring

Key Features

Performance & Scalability

- **In-memory processing** with optional persistence layers
- **Parallel query execution** using thread pools
- **Advanced indexing** with hash, spatial, and semantic indices
- **Optimized graph algorithms** for large-scale analytics
- **Memory-efficient data structures** with automatic cleanup

Enterprise Reliability

- **Schema validation** with type checking and constraints
- **Versioning system** for audit trails and rollback capabilities
- **ACID compliance** for data consistency guarantees
- **Performance monitoring** with detailed metrics collection
- **Comprehensive error handling** with graceful degradation

Advanced Analytics

- **Graph algorithms**: Shortest path, centrality measures, community detection
- **Pattern matching**: Complex relationship queries and traversals
- **Semantic reasoning**: Support for inference and rule-based reasoning
- **Machine learning integration**: Embedding support and feature extraction
- **Real-time analytics**: Streaming updates and incremental computation

Installation and Setup

Requirements

bash

```bash
# Core dependencies
pip install numpy>=1.21.0
pip install pandas>=1.3.0
pip install pydantic>=1.8.0

# Optional ML dependencies
pip install scikit-learn>=1.0.0
pip install networkx>=2.6.0
```

Basic Setup

python

```python
from enterprise_knowledge_graph import (
    KnowledgeGraph, GraphNode, GraphEdge,
    NodeType, EdgeType, GraphSchema
)

# Initialize knowledge graph
config = {
    'max_workers': 8,
    'enable_performance_monitoring': True,
    'cache_size': 10000
}

kg = KnowledgeGraph(config)
```

Quick Start Guide

1. Creating Nodes and Edges

python

```python
import asyncio
import datetime
from enterprise_knowledge_graph import *

async def basic_example():
    # Initialize knowledge graph
    kg = KnowledgeGraph()

    # Create nodes
    person_node = GraphNode(
        id="person_001",
        type=NodeType.PERSON,
        label="John Doe",
        properties={
            "age": 35,
            "occupation": "Data Scientist",
            "location": "San Francisco",
            "skills": ["Python", "Machine Learning", "Graph Theory"]
        }
    )

    company_node = GraphNode(
        id="company_001",
        type=NodeType.ORGANIZATION,
        label="TechCorp Inc",
        properties={
            "industry": "Technology",
            "founded": 2010,
            "employees": 5000,
            "revenue": 1000000000
        }
    )

    project_node = GraphNode(
        id="project_001",
        type=NodeType.ENTITY,
        label="AI Knowledge System",
        properties={
            "status": "active",
            "budget": 2000000,
            "deadline": "2024-12-31"
        }
    )

    # Add nodes to graph
    # Add nodes to graph
    await kg.add_node(person_node)
    await kg.add_node(company_node)
    await kg.add_node(project_node)

    # Create relationships
    works_for_edge = GraphEdge(
        id="edge_001",
        source_id="person_001",
        target_id="company_001",
        type=EdgeType.RELATED_TO,
        label="works_for",
        properties={
            "start_date": "2020-01-15",
            "position": "Senior Data Scientist",
            "salary": 150000
        },
        weight=1.0
    )

    leads_project_edge = GraphEdge(
        id="edge_002",
        source_id="person_001",
        target_id="project_001",
        type=EdgeType.RELATED_TO,
        label="leads",
        properties={
            "role": "Technical Lead",
            "responsibility_level": "high"
        },
        weight=0.9
    )

    # Add edges to graph
    await kg.add_edge(works_for_edge)
    await kg.add_edge(leads_project_edge)

    print(f"Graph created with {len(kg.nodes)} nodes and {len(kg.edges)} edges")

    # Search for nodes
    tech_companies = await kg.search_nodes({
        "type": "organization",
        "industry": "Technology"
    })

    print(f"Found {len(tech_companies)} technology companies")
```

```
        await kg.close()

asyncio.run(basic_example())
```

2. Schema Definition and Validation

python

```python
async def schema_example():
    kg = KnowledgeGraph()

    # Define schema for person nodes
    kg.schema.define_node_schema(
        NodeType.PERSON,
        required_properties=["name", "age"],
        property_types={
            "name": str,
            "age": int,
            "email": str
        },
        constraints=[
            lambda node: node.properties.get("age", 0) >= 0,
            lambda node: "@" in node.properties.get("email", "@")
        ]
    )

    # Define schema for employment edges
    kg.schema.define_edge_schema(
        EdgeType.RELATED_TO,
        allowed_source_types=[NodeType.PERSON],
        allowed_target_types=[NodeType.ORGANIZATION],
        required_properties=["start_date"],
        property_types={
            "start_date": str,
            "salary": int
        }
    )

    # Create valid node
    valid_person = GraphNode(
        id="person_002",
        type=NodeType.PERSON,
        label="Jane Smith",
        properties={
            "name": "Jane Smith",
            "age": 28,
            "email": "jane@example.com"
        }
    )

    await kg.add_node(valid_person)  # Will succeed

    # Try to create invalid node (will raise validation error)
    # Try to create invalid node (will raise validation error)
    try:
        invalid_person = GraphNode(
            id="person_003",
            type=NodeType.PERSON,
            label="Invalid Person",
            properties={
                "name": "Bob",
                "age": -5,  # Invalid age
                "email": "invalid-email"  # Invalid email
            }
        )
        await kg.add_node(invalid_person)
    except ValueError as e:
        print(f"Validation failed: {e}")

    await kg.close()

asyncio.run(schema_example())
```

3. Advanced Querying and Graph Algorithms

python

```python
async def advanced_querying_example():
    kg = KnowledgeGraph()

    # Create a more complex graph
    # ... add nodes and edges (company hierarchy, projects, skills, etc.)

    # Populate sample data
    await create_sample_corporate_graph(kg)

    # Find shortest path between two people
    path = await kg.find_shortest_path("person_001", "person_005")
    print(f"Shortest path: {path}")

    # Calculate centrality measures
    centrality = await kg.calculate_centrality("betweenness")
    most_central = max(centrality.items(), key=lambda x: x[1])
    print(f"Most central node: {most_central[0]} (score: {most_central[1]:.3f})")

    # Detect communities
    communities = await kg.detect_communities("louvain")
    community_sizes = {}
    for node_id, community in communities.items():
        community_sizes[community] = community_sizes.get(community, 0) + 1

    print(f"Detected {len(community_sizes)} communities:")
    for community, size in community_sizes.items():
        print(f"  Community {community}: {size} members")

    # Complex queries
    senior_people = await kg.search_nodes({
        "type": "person",
        "seniority_level": "senior"
    })

    # Find all projects led by senior people
    senior_projects = []
    for person in senior_people:
        outgoing_edges = await kg.get_outgoing_edges(person.id)
        for edge in outgoing_edges:
            if edge.label == "leads" and edge.properties.get("type") == "project":
                project = await kg.get_node(edge.target_id)
                if project:
                    senior_projects.append(project)

    print(f"Projects led by senior people: {len(senior_projects)}")
    print(f"Projects led by senior people: {len(senior_projects)}")

    await kg.close()

async def create_sample_corporate_graph(kg):
    """Create a sample corporate graph structure."""
    # Create employees
    employees = [
        GraphNode(
            id=f"person_{i:03d}",
            type=NodeType.PERSON,
            label=f"Employee {i}",
            properties={
                "department": ["Engineering", "Marketing", "Sales", "HR"][i % 4],
                "seniority_level": ["junior", "mid", "senior"][i % 3],
                "years_experience": i % 10 + 1
            }
        )
        for i in range(1, 21)  # 20 employees
    ]

    # Create departments
    departments = [
        GraphNode(
            id="dept_eng",
            type=NodeType.ORGANIZATION,
            label="Engineering",
            properties={"budget": 5000000, "headcount": 50}
        ),
        GraphNode(
            id="dept_marketing",
            type=NodeType.ORGANIZATION,
            label="Marketing",
            properties={"budget": 2000000, "headcount": 20}
        )
    ]

    # Create projects
    projects = [
        GraphNode(
            id=f"project_{i:03d}",
            type=NodeType.ENTITY,
            label=f"Project {i}",
            properties={
                "status": ["active", "completed", "on_hold"][i % 3],
                "priority": ["high", "medium", "low"][i % 3]
            }
        )
        for i in range(1, 11)  # 10 projects
    ]

    # Add all nodes
    for node in employees + departments + projects:
        await kg.add_node(node)

    # Create relationships
    edge_id = 1

    # Employee -> Department relationships
    for emp in employees:
        dept_id = "dept_eng" if emp.properties["department"] == "Engineering" else "dep
        edge = GraphEdge(
            id=f"edge_{edge_id:03d}",
            source_id=emp.id,
            target_id=dept_id,
            type=EdgeType.PART_OF,
            label="member_of",
            properties={"join_date": "2020-01-01"}
        )
        await kg.add_edge(edge)
        edge_id += 1

    # Employee -> Project relationships
    for i, project in enumerate(projects):
        # Assign 2-3 people per project
        assigned_people = employees[i*2:(i*2)+3]
        for j, person in enumerate(assigned_people):
            edge = GraphEdge(
                id=f"edge_{edge_id:03d}",
                source_id=person.id,
                target_id=project.id,
                type=EdgeType.RELATED_TO,
                label="works_on" if j > 0 else "leads",
                properties={"allocation": 0.5 if j > 0 else 1.0}
            )
            await kg.add_edge(edge)
            edge_id += 1

asyncio.run(advanced_querying_example())
```

4. Graph Analytics and Machine Learning Integration

python

```python
import numpy as np
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA

async def ml_integration_example():
    kg = KnowledgeGraph()

    # Create sample graph with embeddings
    await create_sample_graph_with_embeddings(kg)

    # Extract node embeddings for analysis
    embeddings = []
    node_ids = []

    for node_id, node in kg.nodes.items():
        if node.embedding is not None:
            embeddings.append(node.embedding)
            node_ids.append(node_id)

    if embeddings:
        embeddings_array = np.array(embeddings)

        # Perform clustering on embeddings
        kmeans = KMeans(n_clusters=3, random_state=42)
        clusters = kmeans.fit_predict(embeddings_array)

        # Visualize with PCA
        pca = PCA(n_components=2)
        embeddings_2d = pca.fit_transform(embeddings_array)

        print("Node clustering based on embeddings:")
        for i, (node_id, cluster) in enumerate(zip(node_ids, clusters)):
            node = kg.nodes[node_id]
            print(f"{node.label}: Cluster {cluster}")

        # Find similar nodes using embedding distance
        def find_similar_nodes(target_node_id, top_k=3):
            target_embedding = kg.nodes[target_node_id].embedding
            similarities = []

            for node_id, node in kg.nodes.items():
                if node_id != target_node_id and node.embedding is not None:
                    # Cosine similarity
                    similarity = np.dot(target_embedding, node.embedding) / (
                        np.linalg.norm(target_embedding) * np.linalg.norm(node.embeddi
                        np.linalg.norm(target_embedding) * np.linalg.norm(node.embeddir
                    )
                    similarities.append((node_id, similarity))

            similarities.sort(key=lambda x: x[1], reverse=True)
            return similarities[:top_k]

        # Find nodes similar to the first node
        if node_ids:
            similar = find_similar_nodes(node_ids[0])
            print(f"\nNodes similar to {kg.nodes[node_ids[0]].label}:")
            for node_id, similarity in similar:
                print(f"  {kg.nodes[node_id].label}: {similarity:.3f}")

    await kg.close()

async def create_sample_graph_with_embeddings(kg):
    """Create sample graph with node embeddings."""
    categories = ["technology", "science", "arts", "business"]

    for i in range(20):
        # Generate random embedding
        embedding = np.random.normal(0, 1, 128)

        # Add category bias to embedding
        category = categories[i % len(categories)]
        if category == "technology":
            embedding[:32] += 2.0
        elif category == "science":
            embedding[32:64] += 2.0
        elif category == "arts":
            embedding[64:96] += 2.0
        else:  # business
            embedding[96:] += 2.0

        node = GraphNode(
            id=f"concept_{i:03d}",
            type=NodeType.CONCEPT,
            label=f"{category.title()} Concept {i}",
            properties={
                "category": category,
                "importance": np.random.random()
            },
            embedding=embedding
        )

        await kg.add_node(node)
```

```
asyncio.run(ml_integration_example())
```

Production Deployment

High-Performance Configuration

python

```python
# production_config.py
from enterprise_knowledge_graph import KnowledgeGraph

def create_production_kg():
    """Create production-optimized knowledge graph."""
    config = {
        # Performance settings
        'max_workers': 16,
        'enable_performance_monitoring': True,
        'cache_size': 100000,

        # Memory management
        'memory_limit_gb': 8,
        'gc_threshold': 10000,

        # Persistence settings
        'auto_save_interval': 300,  # 5 minutes
        'backup_retention_days': 30,

        # Indexing strategy
        'indexing_strategy': 'hash',
        'enable_spatial_index': True,
        'enable_text_index': True,

        # Query optimization
        'query_cache_size': 10000,
        'max_query_depth': 20,
        'query_timeout_seconds': 30
    }

    return KnowledgeGraph(config)
```

Persistence and Backup

python

```python
import json
import pickle
from pathlib import Path

class KnowledgeGraphPersistence:
    """Handles persistence and backup for knowledge graphs."""

    def __init__(self, kg: KnowledgeGraph, storage_path: Path):
        self.kg = kg
        self.storage_path = Path(storage_path)
        self.storage_path.mkdir(parents=True, exist_ok=True)

    async def save_to_json(self, filename: str = None):
        """Save graph to JSON format."""
        if filename is None:
            timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
            filename = f"knowledge_graph_{timestamp}.json"

        filepath = self.storage_path / filename
        graph_data = await self.kg.export_to_dict()

        with open(filepath, 'w') as f:
            json.dump(graph_data, f, indent=2, default=str)

        return filepath

    async def load_from_json(self, filepath: Path):
        """Load graph from JSON format."""
        with open(filepath, 'r') as f:
            graph_data = json.load(f)

        await self.kg.import_from_dict(graph_data)
        return True

    async def save_to_pickle(self, filename: str = None):
        """Save graph to pickle format (faster but less portable)."""
        if filename is None:
            timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
            filename = f"knowledge_graph_{timestamp}.pkl"

        filepath = self.storage_path / filename
        graph_data = await self.kg.export_to_dict()

        with open(filepath, 'wb') as f:
            pickle.dump(graph_data, f)
            pickle.dump(graph_data, f)

        return filepath

    async def create_backup(self):
        """Create timestamped backup."""
        timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
        backup_dir = self.storage_path / "backups" / timestamp
        backup_dir.mkdir(parents=True, exist_ok=True)

        # Save in multiple formats
        json_path = await self.save_to_json(str(backup_dir / "graph.json"))
        pickle_path = await self.save_to_pickle(str(backup_dir / "graph.pkl"))

        # Save metadata
        metadata = {
            "created_at": timestamp,
            "statistics": self.kg.get_statistics(),
            "formats": ["json", "pickle"]
        }

        with open(backup_dir / "metadata.json", 'w') as f:
            json.dump(metadata, f, indent=2, default=str)

        return backup_dir

# Example usage
async def persistence_example():
    kg = KnowledgeGraph()

    # Create sample data
    await create_sample_corporate_graph(kg)

    # Setup persistence
    persistence = KnowledgeGraphPersistence(kg, Path("./kg_storage"))

    # Save graph
    json_path = await persistence.save_to_json()
    print(f"Graph saved to: {json_path}")

    # Create backup
    backup_dir = await persistence.create_backup()
    print(f"Backup created: {backup_dir}")

    # Load into new graph
    new_kg = KnowledgeGraph()
    new_persistence = KnowledgeGraphPersistence(new_kg, Path("./kg_storage"))
    await new_persistence.load_from_json(json_path)

    print(f"Loaded graph: {len(new_kg.nodes)} nodes, {len(new_kg.edges)} edges")

    await kg.close()
    await new_kg.close()

asyncio.run(persistence_example())
```

Distributed Processing

python

```python
import concurrent.futures
from typing import List, Callable

class DistributedGraphProcessor:
    """Handles distributed processing of large graphs."""

    def __init__(self, kg: KnowledgeGraph, num_workers: int = 4):
        self.kg = kg
        self.num_workers = num_workers
        self.executor = concurrent.futures.ThreadPoolExecutor(max_workers=num_workers)

    async def parallel_node_processing(
        self,
        processor_func: Callable[[GraphNode], Any],
        batch_size: int = 1000
    ) -> List[Any]:
        """Process nodes in parallel batches."""
        nodes = list(self.kg.nodes.values())
        results = []

        # Process in batches
        for i in range(0, len(nodes), batch_size):
            batch = nodes[i:i + batch_size]

            # Submit batch to thread pool
            futures = [
                self.executor.submit(processor_func, node)
                for node in batch
            ]

            # Collect results
            batch_results = [future.result() for future in futures]
            results.extend(batch_results)

            print(f"Processed batch {i//batch_size + 1}/{(len(nodes)-1)//batch_size +

        return results

    async def parallel_subgraph_analysis(
        self,
        analyzer_func: Callable[[List[GraphNode], List[GraphEdge]], Any],
        partition_size: int = 500
    ) -> List[Any]:
        """Analyze graph partitions in parallel."""
        # Simple partitioning by node ID hash
        # Simple partitioning by node ID hash
        partitions = {}

        for node in self.kg.nodes.values():
            partition_id = hash(node.id) % self.num_workers
            if partition_id not in partitions:
                partitions[partition_id] = {'nodes': [], 'edges': []}
            partitions[partition_id]['nodes'].append(node)

        # Add edges to appropriate partitions
        for edge in self.kg.edges.values():
            source_partition = hash(edge.source_id) % self.num_workers
            partitions[source_partition]['edges'].append(edge)

        # Process partitions in parallel
        futures = [
            self.executor.submit(
                analyzer_func,
                partition['nodes'],
                partition['edges']
            )
            for partition in partitions.values()
        ]

        results = [future.result() for future in futures]
        return results

    def close(self):
        """Clean up thread pool."""
        self.executor.shutdown(wait=True)

# Example usage
async def distributed_processing_example():
    kg = KnowledgeGraph()
    await create_sample_corporate_graph(kg)

    processor = DistributedGraphProcessor(kg, num_workers=4)

    # Example: Extract features from all nodes in parallel
    def extract_features(node: GraphNode) -> Dict[str, Any]:
        return {
            'id': node.id,
            'type': node.type.value,
            'property_count': len(node.properties),
            'has_embedding': node.embedding is not None
        }
    features = await processor.parallel_node_processing(extract_features)
    print(f"Extracted features from {len(features)} nodes")

    # Example: Analyze subgraphs in parallel
    def analyze_subgraph(nodes: List[GraphNode], edges: List[GraphEdge]) -> Dict[str, 
        return {
            'node_count': len(nodes),
            'edge_count': len(edges),
            'density': len(edges) / (len(nodes) * (len(nodes) - 1)) if len(nodes) > 1 
        }

    subgraph_analyses = await processor.parallel_subgraph_analysis(analyze_subgraph)
    print(f"Analyzed {len(subgraph_analyses)} subgraphs")

    processor.close()
    await kg.close()

asyncio.run(distributed_processing_example())
```

Performance Monitoring and Optimization

Metrics Collection

python

```python
class GraphMetrics:
    """Advanced metrics collection for knowledge graphs."""

    def __init__(self, kg: KnowledgeGraph):
        self.kg = kg
        self.start_time = datetime.datetime.now()

    def get_comprehensive_metrics(self) -> Dict[str, Any]:
        """Get comprehensive performance metrics."""
        stats = self.kg.get_statistics()

        # Calculate additional metrics
        uptime = (datetime.datetime.now() - self.start_time).total_seconds()

        # Graph structure metrics
        avg_degree = (2 * stats['edges']) / max(1, stats['nodes'])
        density = (2 * stats['edges']) / max(1, stats['nodes'] * (stats['nodes'] - 1))

        # Performance metrics
        queries_per_second = stats['metrics']['queries_executed'] / max(1, uptime)

        return {
            **stats,
            'uptime_seconds': uptime,
            'average_degree': avg_degree,
            'graph_density': density,
            'queries_per_second': queries_per_second,
            'memory_efficiency': stats['memory_usage_bytes'] / max(1, stats['nodes'] +
        }

    async def benchmark_operations(self, iterations: int = 1000) -> Dict[str, float]:
        """Benchmark common graph operations."""
        import time

        # Node addition benchmark
        start_time = time.time()
        for i in range(iterations):
            node = GraphNode(
                id=f"benchmark_node_{i}",
                type=NodeType.ENTITY,
                label=f"Benchmark Node {i}",
                properties={"benchmark": True}
            )
            await self.kg.add_node(node)
        node_addition_time = (time.time() - start_time) / iterations

        # Query benchmark
        start_time = time.time()
        for i in range(iterations):
            await self.kg.search_nodes({"benchmark": True}, limit=10)

        query_time = (time.time() - start_time) / iterations

        # Cleanup benchmark nodes
        benchmark_nodes = await self.kg.search_nodes({"benchmark": True})
        for node in benchmark_nodes:
            await self.kg.remove_node(node.id)

        return {
            'node_addition_ms': node_addition_time * 1000,
            'query_ms': query_time * 1000,
            'operations_per_second': 1 / max(node_addition_time, 1e-6)
        }

# Example usage
async def monitoring_example():
    kg = KnowledgeGraph()
    await create_sample_corporate_graph(kg)

    metrics_collector = GraphMetrics(kg)

    # Get comprehensive metrics
    metrics = metrics_collector.get_comprehensive_metrics()
    print("Graph Metrics:")
    for key, value in metrics.items():
        if isinstance(value, float):
            print(f"  {key}: {value:.3f}")
        else:
            print(f"  {key}: {value}")

    # Run performance benchmark
    benchmark_results = await metrics_collector.benchmark_operations(100)
    print("\nBenchmark Results:")
    for operation, time_ms in benchmark_results.items():
        print(f"  {operation}: {time_ms:.3f}")

    await kg.close()

asyncio.run(monitoring_example())
```

Best Practices

1. Schema Design

- **Define schemas early** for data consistency and validation
- **Use appropriate node types** to enable type-specific queries
- **Implement constraints** to maintain data quality
- **Version your schemas** for backward compatibility

2. Performance Optimization

- **Use appropriate indexing** strategies for your query patterns
- **Batch operations** when adding large amounts of data
- **Monitor memory usage** and implement cleanup procedures
- **Optimize query patterns** by analyzing execution times

3. Data Modeling

- **Keep properties lightweight** to improve memory efficiency
- **Use embeddings** for semantic similarity and ML integration
- **Model temporal relationships** explicitly when needed
- **Implement proper versioning** for audit trails

4. Production Deployment

- **Implement proper backup strategies** with multiple formats
- **Monitor performance metrics** continuously
- **Use distributed processing** for large-scale analytics
- **Implement proper error handling** and graceful degradation

Integration Patterns

Database Integration

python

```python
# Example: Sync with relational database
async def sync_with_database(kg: KnowledgeGraph, db_connection):
    """Sync knowledge graph with relational database."""

    # Load entities from database
    entities = await db_connection.fetch("SELECT * FROM entities")

    for entity in entities:
        node = GraphNode(
            id=f"entity_{entity['id']}",
            type=NodeType.ENTITY,
            label=entity['name'],
            properties={
                'db_id': entity['id'],
                'created_at': entity['created_at'],
                'updated_at': entity['updated_at']
            }
        )
        await kg.add_node(node)

    # Load relationships
    relationships = await db_connection.fetch("SELECT * FROM relationships")

    for rel in relationships:
        edge = GraphEdge(
            id=f"rel_{rel['id']}",
            source_id=f"entity_{rel['source_id']}",
            target_id=f"entity_{rel['target_id']}",
            type=EdgeType.RELATED_TO,
            label=rel['relationship_type'],
            properties={'db_id': rel['id']}
        )
        await kg.add_edge(edge)
```

API Integration

python

```python
# Example: REST API for knowledge graph
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI()
kg = KnowledgeGraph()

class NodeCreate(BaseModel):
    id: str
    type: str
    label: str
    properties: Dict[str, Any] = {}

class EdgeCreate(BaseModel):
    id: str
    source_id: str
    target_id: str
    type: str
    label: str
    properties: Dict[str, Any] = {}

@app.post("/nodes")
async def create_node(node_data: NodeCreate):
    try:
        node = GraphNode(
            id=node_data.id,
            type=NodeType(node_data.type),
            label=node_data.label,
            properties=node_data.properties
        )
        await kg.add_node(node)
        return {"status": "success", "id": node.id}
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))

@app.get("/nodes/{node_id}")
async def get_node(node_id: str):
    node = await kg.get_node(node_id)
    if not node:
        raise HTTPException(status_code=404, detail="Node not found")
    return node.to_dict()

@app.get("/search/nodes")
async def search_nodes(query: str, limit: int = 100):
    # Parse query string into search parameters
    # Parse query string into search parameters
    search_params = {"label": query}  # Simplified
    results = await kg.search_nodes(search_params, limit)
    return [node.to_dict() for node in results]
```

License

This Knowledge Graph Engine is released under the MIT License, making it suitable for both commercial and open-source AI projects.

Contributing

Contributions are welcome! Areas of particular interest:

- Additional graph algorithms
- New indexing strategies
- Performance optimizations
- Integration patterns
- Documentation improvements

For questions or support, please refer to the project documentation or open an issue in the repository.