Tree Versioning System - Python Take Home Assignment

Background

Our system manages tree structures in a SQL database with three main tables. Each tree can represent a configuration hierarchy, where nodes contain configuration data and edges define relationships with additional metadata.

Current Schema

```
Unset
CREATE TABLE Tree (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
Unset
CREATE TABLE TreeNode (
    id INTEGER PRIMARY KEY,
    tree_id INTEGER NOT NULL,
    data JSON,  -- Stores node-specific data
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (tree_id) REFERENCES Tree(id)
);
```

```
Unset
CREATE TABLE TreeEdge (
    id INTEGER PRIMARY KEY,
    incoming_node_id INTEGER NOT NULL,  # origin node for this
edge
```

```
    outgoing_node_id INTEGER NOT NULL,  # destination node for
this edge
    data JSON,  -- Stores edge-specific metadata
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (incoming_node_id) REFERENCES TreeNode(id),
    FOREIGN KEY (outgoing_node_id) REFERENCES TreeNode(id)
);
```

Example Use Cases

Python

```
# Configuration Management


# Creating a new configuration version
tree = Tree.get(id=1)
new_tag = tree.create_tag("release-v1.0", description="Initial
stable release")

# Making changes to the configuration
modified_tree =
tree.create_new_tree_version_from_tag("release-v1.0")
new_node = modified_tree.add_node(data={"setting": "new_value"})
modified_tree.create_tag("release-v1.1", description="Added new
setting")

# Adding an edge between nodes
modified_tree.add_edge(node_id_1=1, node_id_2=2, data={"weight":
0.5})

# Retrieving historical configuration
old_config = tree.get_by_tag("release-v1.0")
```

Python

```python
# Feature Branching

# Creating a feature branch
main_tree = Tree.get_by_tag("main-v2.0")
feature_branch =
main_tree.create_new_tree_version_from_tag("main-v2.0")

# Making feature-specific changes
node1 = feature_branch.add_node(data={"feature_flag": True})
node2 = feature_branch.add_node(data={"config": "new_setting"})
feature_branch.add_edge(node_id_1=node1.id, node_id_2=node2.id,
data={"relation": "depends_on"})
feature_branch.create_tag("feature-x-v1", description="Feature X
implementation")
```

Python

```python
# Rollback Scenario

# Marking a known good state
stable_tree = Tree.get(id=1)
stable_tag = stable_tree.create_tag("stable-v1")

# Making potentially risky changes
new_node = stable_tree.add_node(data={"experimental": True})
stable_tree.add_edge(node_id_1=1, node_id_2=new_node.id,
data={"type": "experimental"})

# Rolling back if issues are found
if problems_detected():
    rollback_tree = stable_tree.restore_from_tag("stable-v1")
```

Python

```python
# Tree Fetching by tag

# Get a tree by its tag
historical_tree = Tree.get_by_tag("release-v1.0")

# Get the root node(s)
root_node = historical_tree.get_root_node()

# Traverse from a specific node
node = historical_tree.get_node(node_id=1)
children = historical_tree.get_child_nodes(node_id=1)
parents = historical_tree.get_parent_nodes(node_id=1)

# Get edge information
edges = historical_tree.get_node_edges(node_id=1)
for edge in edges:
    print(f"Edge {edge.id} metadata: {edge.data}")
    print(f"Connected to node: {edge.outgoing_node_id}")

# Traverse entire tree structure
def traverse_tree(tree, node_id):
    node = tree.get_node(node_id)
    print(f"Node {node_id} metadata: {node.data}")

    edges = tree.get_node_edges(node_id)
    for edge in edges:
        print(f"Edge metadata: {edge.data}")
        traverse_tree(tree, edge.outgoing_node_id)

# Start traversal from root
for root in historical_tree.get_root_nodes():
    traverse_tree(historical_tree, root.id)

# Get all nodes at a specific depth
level_2_nodes = historical_tree.get_nodes_at_depth(2)
```

```
# Find path between nodes
path = historical_tree.find_path(start_node_id=1, end_node_id=5)
for node, edge in path:
    print(f"Node: {node.data}")
    print(f"Connected by edge: {edge.data if edge else 'None'}")
```

Assignment Overview

Design and implement a versioning system that allows users to:
1. Tag specific tree configurations with meaningful labels
2. Create new versions from existing tagged configurations
3. Manage and navigate between different versions of the same tree
4. Traverse and inspect tree structures at any tagged point in time

Technical Requirements

Required Technologies

1. Python 3.9+
2. SQL database of your choice
3. Testing framework of your choice

Core Features

1. Implement a tagging system that can:
   - Mark specific tree configurations with user-defined tags
   - Store metadata about when the tag was created
   - Support retrieval of tree structure at any tagged point

2. Implement version management that:
   - Allows creation of new versions from any tagged configuration
   - Maintains relationship between parent and child versions

3. Implement tree traversal and inspection that:
   - Allows accessing exact state of nodes and edges at any tagged point

Detailed Deliverables

Please provide your solution as a Git repository with:

1. Database Implementation
- Models for all necessary tables
- Migration scripts for schema changes
- Appropriate indexes for efficient querying

2. Core Implementation in python code
The code should be runnable as python code (possibly connected to something as simple as a local SQLite instance) (UI is not a requirement)

3. Tests
- Unit tests for all core functionality
- Integration tests for database operations

4. Documentation
- Example usage scenarios
- Installation and setup guide
- Design decisions and tradeoffs