# CognitiveGraph Enhancements: Technical Design Specification

Version: 1.2

Date: 2025-10-22

Status: FINAL - REVISED

## 1. Introduction

This document specifies the required changes to the CognitiveGraph library to add support for large-scale persistence, efficient querying, and a graph mutation API. The goals are to enable consumers like the Golem GSSM and the StepParserEngine to handle TB-scale datasets, find nodes by source location, and create modified versions of the graph.

## 2. Design: Large-Scale Persistence & Disk Mapping

* **Goal:** To enable CognitiveGraph to transparently handle graph data that exceeds available RAM, as required for TB-scale GSSM training data.
* **Architecture:** The library will use memory-mapped files to provide zero-copy access to graph data stored on disk. A caching layer will be used to keep frequently accessed data ("hot" nodes) in memory for optimal performance.

### 2.1. CognitiveGraphBuilder Enhancements

* **Change:** The Build() method will be overloaded to support writing directly to a file stream.
* **Implementation:**
  + A new overload, public void Build(FileStream stream, ...) will be added.
  + This method will write all graph data (header, nodes, properties, indices) directly to the provided file stream, avoiding the need to hold the entire buffer in memory before saving.

### 2.2. CognitiveGraph Accessor Enhancements

* **Change:** The CognitiveGraph constructor will be overloaded to accept a file path.
* **Implementation:**
  + A new constructor, public CognitiveGraph(string filePath), will be added.
  + Internally, it will use System.IO.MemoryMappedFiles.MemoryMappedFile to create a view accessor over the on-disk graph file.
  + The readonly ref struct accessors (SymbolNode, Property, etc.) will then operate on this memory-mapped view, providing the same zero-copy performance as the in-memory version but without loading the entire file into RAM.

### 2.3. Caching Layer for Hot Data

* **Change:** A caching mechanism will be integrated to improve performance for disk-backed graphs.
* **Implementation:**
  + The CognitiveGraph class will internally use an IMemoryCache instance.
  + When a node or property is accessed frequently, its deserialized data or the accessor struct itself can be stored in the cache.
  + This avoids the I/O overhead of repeatedly accessing the memory-mapped file for "hot" parts of the graph, directly addressing the caching strategy mentioned in the GSSM document.
* **Affected Files:** CognitiveGraph.cs, Builder/CognitiveGraphBuilder.cs, Buffer/CognitiveGraphBuffer.cs.

## 3. Design: Interval Tree Index

* **Goal:** To provide efficient spatial querying (finding nodes by source code location).
* **Implementation:** A serializable IntervalTree data structure will be implemented and stored in a dedicated section of the CognitiveGraphBuffer, indexed by the GraphHeader. The public API will expose a FindNodesAt(uint byteOffset) method that uses this index for fast lookups.
* **Affected Files:** CognitiveGraph.cs, Builder/CognitiveGraphBuilder.cs, Schema/GraphHeader.cs, and a new IntervalTree.cs file.

## 4. Design: High-Performance GraphQL Query Engine

* **Goal:** To implement the Query(string query) placeholder with a powerful, standard query language while adhering to the library's low-allocation, zero-copy principles.
* **Architecture:** A custom execution visitor will be built on top of the GraphQL-dotnet parser. This visitor will traverse the GraphQL query AST and translate it into operations against the high-performance, zero-copy SymbolNode accessors.
* **Benefit:** This provides the convenience of GraphQL syntax while returning a collection of SymbolNode "pointers", avoiding the performance penalty of allocating new result objects.
* **Affected Files:** CognitiveGraph.cs, and new files for the GraphQL integration, custom visitor, and a fluent C# query API.

## 5. Design: High-Performance Graph Mutation API

* **Goal:** To provide a formal, efficient, and user-friendly API for creating modified versions of an existing CognitiveGraph, based on the proven pattern in Minotaur/migration\_to\_cognitive\_graph.
* **Architecture:** Since the CognitiveGraphBuffer is immutable, mutations must produce a new graph. The API will be designed around an **editor class** that efficiently reconstructs the graph while applying a batch of edit operations. This formalizes the "rebuild" pattern and moves it from the consuming application (Minotaur) into the core CognitiveGraph library.

### 5.1. CognitiveGraphEditor Class

* **Change:** A new public class, CognitiveGraphEditor, will be added to the library.
* **Implementation:**
  + The editor will be initialized with an existing CognitiveGraph instance (the source). Internally, it will also create a new CognitiveGraphBuilder instance (the destination).
  + It will expose high-level mutation methods: InsertNode, ReplaceNode, DeleteNode, MoveNode, UpdateProperty. These methods will not execute immediately but will instead queue an EditOperation data object.
  + A final CognitiveGraph Build() method will process the entire queue of operations. It will traverse the source graph and write to the destination builder, applying the queued modifications as it goes.
  + **Optimization:** The traversal will be optimized. When encountering a large branch of the source graph that is unaffected by any queued edits, the editor can perform a highly efficient, low-level bulk copy of the corresponding bytes from the source buffer to the destination builder.
  + The Build() method will return a new, modified CognitiveGraph instance.

### 5.2. Transactional Edits

* **Benefit:** The queuing mechanism naturally provides transactional or batch editing. Users can apply multiple changes to the editor and then generate the final result with a single, efficient rebuild operation, which is much faster than rebuilding the graph after each individual edit.
* **Affected Files:** New files for CognitiveGraphEditor.cs and EditOperation.cs will be added to the CognitiveGraph project, migrating the logic from the Minotaur codebase.

## 6. Testing Strategy

* **Change:** Create new test suites in the CognitiveGraph.Tests project to validate the correctness and performance of the persistence layer, spatial index, query engines, and the mutation API.
* **Implementation:**
  1. **Unit Tests (Persistence):**
     + **Large-File Tests:** Create tests that generate and operate on CognitiveGraph files larger than available RAM. Verify correct operation without OutOfMemoryException.
     + **Data Integrity:** Verify graphs serialized to disk and loaded back are identical.
     + **Caching:** Test that the caching layer improves performance for repeated access to disk-mapped graphs.
  2. **Unit Tests (Spatial Index):**
     + Test the IntervalTree structure.
     + Test index serialization and deserialization.
     + Test FindNodesAt() queries.
  3. **Unit Tests (Query Engines):**
     + Test the Query() method with various GraphQL queries.
     + Test the fluent C# API methods.
  4. **Unit Tests (Mutation API):**
     + Test each CognitiveGraphEditor operation (Insert, Delete, etc.).
     + Test batching multiple edits.
     + Test bulk copy optimization.
  5. **Performance Benchmarks:**
     + Measure performance of in-memory vs. disk-mapped queries (with/without cache).
     + Measure performance of GraphQL vs. fluent API queries.
     + Measure performance of graph mutations (single vs. batch).
* **Affected Files:** New test files within the CognitiveGraph.Tests project.

## 7. Review Status

* [x] **Section 2: Design (Persistence)** - **Approved**
* [x] **Section 3: Design (Spatial Index)** - **Approved**
* [x] **Section 4: Design (Query Engine)** - **Approved**
* [x] \*\*Section 5: