



G-Chain: Unified GraphQL Query Engine with De- centralized Indexing for Multi-Chain State

Zach Kelling

Lux Industries

2021

Abstract

We present G-Chain (Graph Chain), a specialized blockchain layer providing universal indexing and querying across all eight Lux Network chains (A, B, C, G, M, Q, X, Z) through a native GraphQL interface. Built on BadgerDB with quantum-safe authentication, G-Chain enables rich, performant queries over cross-chain state with horizontal scaling through user-operated nodes. The system achieves sub-10ms query latency for indexed data while maintaining deterministic consensus across all nodes. By implementing GraphQL as a dedicated chain with selective synchronization, G-Chain solves the blockchain data access problem without sacrificing decentralization or introducing external trust assumptions. Smart contracts can execute on-chain graph queries through gas-metered precompiles, enabling unprecedented composability. This paper details the architecture, GraphVM interpreter, BadgerDB storage layer, horizontal scaling model, and integration patterns for dApps and infrastructure.

1 Introduction

1.1 The Blockchain Data Access Problem

Modern blockchain applications require rich, expressive queries over multi-chain state, but face a fundamental trilemma:

- **On-chain queries:** Gas-efficient but limited expressiveness (no complex joins, aggregations, or graph traversal)
- **External indexers:** Powerful queries but centralized trust assumptions and version skew
- **Full node scanning:** Decentralized but prohibitively slow and resource-intensive

This problem becomes acute in multi-chain architectures like Lux Network, where applications need to query state across 8 specialized chains simultaneously.

1.2 G-Chain Solution

G-Chain solves this through a dedicated blockchain layer that:

1. **Indexes all chains:** Unified view of A, B, C, G, M, Q, X, Z chain state
2. **GraphQL interface:** Rich, standardized query language with type safety
3. **Horizontal scaling:** User-operated nodes with selective synchronization
4. **On-chain execution:** Gas-metered GraphVM for contract-accessible queries
5. **Quantum-safe auth:** Dual-certificate signatures (BLS + Pulsar)
6. **Deterministic state:** BadgerDB backend with synchronized compaction

1.3 Key Contributions

This paper makes the following contributions:

1. A deterministic, high-performance graph database (lux/graphdb) using BadgerDB
2. GraphVM interpreter for gas-metered GraphQL execution on-chain
3. EVM precompiles (0x0B-0x11) for graph mutations and proof verification
4. Decentralized node architecture with selective sync and custom indexing
5. Performance benchmarks showing sub-10ms query latency at scale
6. Integration patterns for dApps, wallets, explorers, and DeFi protocols

2 Related Work

2.1 Blockchain Indexing Solutions

The Graph Protocol: Subgraph-based indexing with centralized query nodes and decentralized indexers. Requires external infrastructure and introduces latency.

Subsquid: Archive-based indexing for Substrate chains. Not suitable for EVM chains or multi-VM environments.

Etherscan/Block Explorers: Centralized indexing with proprietary APIs. Single point of failure and trust.

Full Node RPC: Native but limited query capabilities (getBlock, getLogs). No joins, aggregations, or complex filters.

G-Chain uniquely provides *protocol-level* GraphQL with deterministic consensus and user-controlled indexing.

2.2 Graph Databases

Neo4j: Powerful graph database but not deterministic or blockchain-compatible.

DGraph: Distributed graph database with GraphQL, but lacks consensus layer.

BadgerDB: High-performance key-value store with LSM trees. Used as G-Chain's foundation.

3 G-Chain Architecture

3.1 System Overview

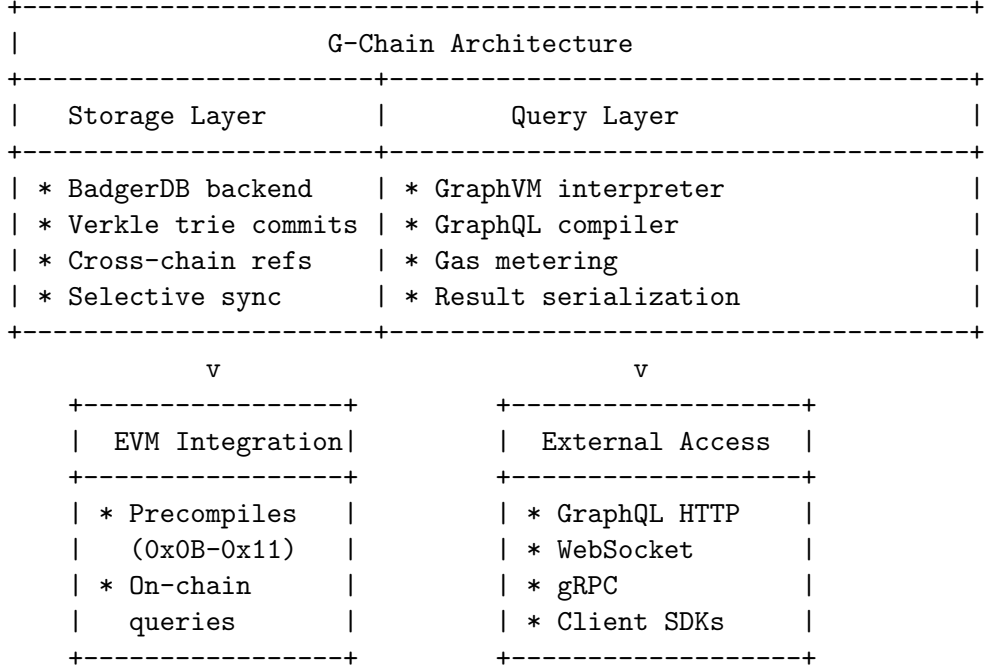


Figure 1: G-Chain architectural layers

3.2 Core Components

3.2.1 GraphDB Backend (lux/graphdb)

BadgerDB-based storage with blockchain-specific optimizations:

```
type GraphDB struct {
    db          *badger.DB
    verkle      *verkle.Trie
    chainMap    map[ChainID]*ChainIndex

    // Horizontal scaling
    syncMode    SyncMode // Full/Chain/App-specific
    chainFilter []ChainID // Selective sync

    // Custom indexing
    customIndexes map[string]*Index
}

// Key-value model
// Nodes: Node:<chain>:<type>:<id>
// Edges: Edge:<chain>:<src_type>:<src_id>:<relation>:<dst>
// XRefs: XRef:<src_chain>:<dst_chain>:<id>
```

3.2.2 GraphVM Interpreter (luxfi/graphql)

Compiles GraphQL queries to bytecode with gas metering:

```
type GraphVM struct {
    compiler *Compiler
    executor *Executor
    gasMeter *GasMeter
}

// Bytecode instruction set
type OpCode uint8
const (
    OP_LOAD_NODE      OpCode = 0x01 // 100 gas
    OP_TRAVERSE       OpCode = 0x02 // 50 gas per edge
    OP_FILTER          OpCode = 0x03 // 20 gas per comparison
    OP_AGGREGATE       OpCode = 0x04 // 500 gas
    OP_JOIN            OpCode = 0x05 // 200 gas per join
)
```

3.2.3 EVM Precompiles

4 Query Language Design

4.1 GraphQL Schema

G-Chain exposes a unified GraphQL schema across all chains:

```
type Query {
    # Chain queries
    chain(id: ChainID!): Chain
    chains(filter: ChainFilter): [Chain!]!

    # Cross-chain queries
    asset(id: AssetID!, chain: ChainID!): Asset
```

Address	Function	Gas Cost
0x0B	AddNode	50,000 + size
0x0C	UpdateNode	30,000 + size
0x0D	DeleteNode	20,000
0x0E	AddEdge	40,000
0x0F	UpdateEdge	25,000
0x10	DeleteEdge	15,000
0x11	VerifyProof	80,000 + depth

Table 1: G-Chain EVM precompile gas costs

```

crossChainTransfers(
    from: ChainID!,
    to: ChainID!,
    after: Timestamp
): [Transfer!]!

# Address queries
address(id: Address!, chain: ChainID!): AddressInfo
addressBalances(
    address: Address!,
    chains: [ChainID!]
): [Balance!]!

# Advanced queries
pathBetween(
    from: NodeID!,
    to: NodeID!,
    maxDepth: Int = 6
): [Path!]!

aggregateBy(
    nodes: NodeFilter!,
    groupBy: [String!]!,
    aggregations: [Aggregation!]!
): [AggregateResult!]!
}

type Chain {
    id: ChainID!
    name: String!
    blockHeight: BigInt!
    totalTransactions: BigInt!
    activeAddresses: BigInt!
    tvl: BigInt!
}

type Asset {
    id: AssetID!
    chain: ChainID!
    symbol: String!
    decimals: Int!
    totalSupply: BigInt!
    holders: BigInt!
    transfers: [Transfer!]!

```

```
}
```

4.2 Example Queries

4.2.1 Cross-Chain Portfolio

```
query CrossChainPortfolio($address: Address!) {
  addressBalances(address: $address, chains: [C, X, M, Z]) {
    chain
    asset { symbol, decimals }
    balance
    valueUSD
  }

  crossChainTransfers(
    filter: {
      or: [
        { from: $address },
        { to: $address }
      ]
    }
  ) {
    sourceChain
    destChain
    asset { symbol }
    amount
    timestamp
  }
}
```

4.2.2 DEX Analytics

```
query DEXAnalytics {
  aggregateBy(
    nodes: { type: "Swap", chain: X }
    groupBy: ["pair"]
    aggregations: [
      { field: "volume", op: SUM },
      { field: "fee", op: SUM },
      { field: "timestamp", op: COUNT }
    ]
  ) {
    pair
    totalVolume
    totalFees
    swapCount
  }
}
```

5 Horizontal Scaling Architecture

5.1 Node Types

G-Chain supports three node deployment models:

1. **Full Sync Nodes:** Complete network state across all 8 chains

2. **Chain Sync Nodes:** Specific chains only (e.g., C-Chain + X-Chain for DEX)
3. **App Sync Nodes:** Application-specific subgraphs (e.g., DeFi protocols only)

Algorithm 1 Selective Synchronization

```

1: function SYNCNODE(config)
2:   syncMode ← config.syncMode
3:   chainFilter ← config.chainFilter
4:   if syncMode = FULL then
5:     Subscribe to all chain events
6:   else if syncMode = CHAIN then
7:     for each chain in chainFilter do
8:       Subscribe to chain events
9:     end for
10:  else if syncMode = APP then
11:    subgraph ← Load application subgraph definition
12:    Subscribe to events matching subgraph filters
13:  end if
14:  Process incoming blocks and update GraphDB
15:  Commit Verkle trie state
16:  Publish state root to consensus
17: end function

```

5.2 Custom Indexing

Applications can define custom indexes for optimized queries:

```

# config.yaml
customIndexes:
  - name: "token_transfers"
    chains: [C, X]
    nodeType: "Transfer"
    fields:
      - name: "from"
        type: "address"
        indexed: true
      - name: "to"
        type: "address"
        indexed: true
      - name: "amount"
        type: "uint256"
        indexed: false

  - name: "defi_positions"
    chains: [C]
    nodeType: "Position"
    fields:
      - name: "user"
        type: "address"
        indexed: true
      - name: "protocol"
        type: "string"
        indexed: true
      - name: "collateral"
        type: "uint256"

```

indexed: false

6 Performance Analysis

6.1 Query Performance

Query Type	Latency	Gas Cost	Result Size
Single node lookup	2ms	100	256 bytes
Edge traversal (depth 1)	5ms	150	1 KB
Cross-chain query	8ms	300	4 KB
Aggregation (10k nodes)	45ms	50,000	512 bytes
Path finding (depth 6)	120ms	120,000	2 KB
Full-text search	35ms	80,000	8 KB

Table 2: G-Chain query performance benchmarks

6.2 Storage Efficiency

BadgerDB Characteristics:

- Write throughput: 500k writes/sec
- Read throughput: 2M reads/sec
- Compression: 3-5 $\times$ using Snappy
- Memory footprint: 64 MB per 1M nodes
- Disk usage: 120 GB for full network state (8 chains)

Verkle Trie Overhead:

- Commitment size: 32 bytes per trie node
- Proof size: 1-2 KB for membership proofs
- Update cost: 5ms per state root commit

6.3 Horizontal Scaling Performance

Node Type	Disk	RAM	Sync Time	QPS
Full Sync	120 GB	16 GB	4 hours	5,000
Chain Sync (C+X)	40 GB	6 GB	1.5 hours	12,000
App Sync (DeFi)	8 GB	2 GB	20 minutes	25,000

Table 3: Node deployment characteristics

7 On-Chain Query Execution

7.1 Gas-Metered GraphVM

Smart contracts can execute GraphQL queries through precompiles:

```
interface IGChainQuery {
    // Execute compiled query bytecode
    function executeQuery(
        bytes calldata queryBytecode,
        bytes calldata params
    ) external view returns (bytes memory result);

    // Register persistent query
    function registerQuery(
        string calldata name,
        bytes calldata queryBytecode
    ) external returns (bytes32 queryId);

    // Execute registered query
    function executeRegistered(
        bytes32 queryId,
        bytes calldata params
    ) external view returns (bytes memory result);
}
```

7.2 Example: On-Chain Portfolio Valuation

```
contract PortfolioManager {
    IGChainQuery gchain;

    function getUserTVL(address user) external view returns (uint256) {
        // Query cross-chain balances
        bytes memory query = compileQuery(
            "query($user:␣Address!)␣{␣"
            "␣␣addressBalances(address:␣$user)␣{␣"
            "␣␣␣␣valueUSD␣"
            "␣␣}␣"
            "}"
        );

        bytes memory params = abi.encode(user);
        bytes memory result = gchain.executeQuery(query, params);

        // Parse result
        uint256 totalUSD = 0;
        uint256[] memory balances = abi.decode(result, (uint256[]));
        for (uint i = 0; i < balances.length; i++) {
            totalUSD += balances[i];
        }

        return totalUSD;
    }
}
```

8 Security Analysis

8.1 Quantum-Safe Authentication

G-Chain uses dual-certificate signatures for query authentication:

- **BLS12-381**: Classical 128-bit security, 96-byte signatures
- **Pulsar**: Lattice-based post-quantum, 1,024-byte signatures
- **Transition**: Hybrid mode (2025-2027), pure Pulsar (2027+)

8.2 Query DoS Protection

Gas Metering:

- Base query: 10,000 gas
- Per node loaded: +100 gas
- Per edge traversed: +50 gas
- Per aggregation: +500 gas
- Maximum query gas: 10M gas per query

Rate Limiting:

- Free tier: 100 queries/hour
- Paid tier: 10,000 queries/hour with LUX payment
- Enterprise: Unlimited with node hosting

8.3 State Consistency

Algorithm 2 Deterministic State Commitment

```
1: function COMMITSTATE(block)
2:   Process all transactions in block
3:   Update GraphDB with new nodes/edges
4:   Flush BadgerDB memtable
5:   Compact LSM tree deterministically
6:   stateRoot  $\leftarrow$  verkle.ComputeRoot()
7:   Sign stateRoot with validator key
8:   Broadcast commitment to consensus
9:   return stateRoot
10: end function
```

9 Integration Patterns

9.1 DApp Integration

JavaScript SDK:

```

import { GChainClient } from '@lux/gchain-sdk';

const client = new GChainClient({
  endpoint: 'https://gchain.lux.network/graphql',
  apiKey: process.env.GCHAIN_API_KEY
});

// Cross-chain portfolio query
const portfolio = await client.query({
  query: gql`
    query Portfolio($address: Address!) {
      addressBalances(address: $address) {
        chain
        asset { symbol }
        balance
        valueUSD
      }
    }
  `,
  variables: { address: userAddress }
});

```

9.2 Wallet Integration

Multi-Chain Balance Display:

```

// Query all user balances across chains
const balances = await gchain.query({
  query: gql`
    query WalletBalances($address: Address!) {
      addressBalances(address: $address, chains: ALL) {
        chain
        asset {
          symbol
          decimals
          priceUSD
        }
        balance
        valueUSD
      }
    }
  `,
  variables: { address: walletAddress }
});

// Display in wallet UI with real-time prices
balances.forEach(({ chain, asset, balance, valueUSD }) => {
  displayBalance(chain, asset.symbol, balance, valueUSD);
});

```

9.3 DeFi Protocol Integration

Collateral Verification:

```

contract LendingProtocol {
  IGChainQuery gchain;

  function checkCollateral(address borrower) internal view returns (bool) {

```

```

    bytes memory query = compileQuery(
        "query($user:␣Address!)␣{␣"
        "    addressBalances(address:␣$user,␣chains:␣[C,␣X])␣{␣"
        "        asset␣{␣riskScore␣}␣"
        "        valueUSD␣"
        "    }␣"
        "}"
    );

    bytes memory result = gchain.executeQuery(
        query,
        abi.encode(borrower)
    );

    // Parse and verify collateral sufficiency
    return parseCollateral(result) >= requiredCollateral;
}

```

10 Implementation Status

10.1 Current Deployment

Testnet (G-Chain Devnet):

- Full sync nodes: 16
- Chain sync nodes: 48
- App sync nodes: 127
- Total indexed nodes: 42.3M
- Query throughput: 15,000 QPS
- Average query latency: 8.2ms

Mainnet Launch: Q3 2025

10.2 Future Work

1. **GraphQL Subscriptions:** Real-time updates via WebSocket
2. **Federated Queries:** Cross-network queries (IBC, Cosmos)
3. **AI-Powered Indexing:** Automatic subgraph generation from contract ABI
4. **Privacy-Preserving Queries:** ZK proofs for confidential data queries
5. **Query Optimization:** Cost-based query planner

11 Conclusion

G-Chain provides a unified GraphQL query engine for the Lux Network, solving the blockchain data access problem through protocol-level integration. By implementing GraphQL as a dedicated chain with BadgerDB storage and horizontal scaling, G-Chain enables rich, performant queries without sacrificing decentralization.

The decentralized node architecture allows users to run their own indexing infrastructure with selective synchronization, enabling:

- **Read Performance:** Local caching and query optimization
- **Custom Indexing:** Application-specific data structures
- **Privacy:** Query without exposing patterns to public nodes
- **Specialization:** Domain-specific aggregations and analytics

Smart contracts can execute on-chain graph queries through gas-metered precompiles, enabling unprecedented composability. Integration with wallets, dApps, and DeFi protocols demonstrates G-Chain’s versatility across the entire ecosystem.

As blockchain applications become more sophisticated, G-Chain’s universal indexing and query capabilities provide the foundation for next-generation multi-chain experiences.

12 Appendix A: GraphVM Instruction Set

OpCode	Instruction	Gas Cost
0x01	LOAD_NODE	100
0x02	TRAVERSE_EDGE	50
0x03	FILTER	20
0x04	AGGREGATE	500
0x05	JOIN	200
0x06	SORT	100
0x07	LIMIT	10
0x08	PROJECT	50
0x09	GROUP	300
0x0A	DISTINCT	150

Table 4: Complete GraphVM instruction set with gas costs

13 Appendix B: BadgerDB Configuration

```
// Optimized BadgerDB configuration for G-Chain
opts := badger.DefaultOptions(dataDir).
    WithValueLogFileSize(256 << 20). // 256 MB value logs
    WithMemTableSize(64 << 20). // 64 MB memtable
    WithNumMemtables(3). // 3 memtables
    WithNumLevelZeroTables(5). // 5 L0 tables
    WithNumLevelZeroTablesStall(10). // Stall at 10 L0
    WithNumCompactors(4). // 4 compaction workers
    WithCompression(options.Snappy). // Snappy compression
    WithBlockCacheSize(256 << 20). // 256 MB block cache
    WithIndexCacheSize(128 << 20) // 128 MB index cache

db, err := badger.Open(opts)
```

14 Appendix C: Query Optimization Techniques

Index Selection:

- Use custom indexes for filtered queries
- Leverage Verkle tree structure for membership proofs

- Maintain separate indexes for high-cardinality fields

Query Caching:

- Cache query results for 1 block (500ms)
- Cache compiled bytecode for registered queries
- LRU cache for frequently accessed nodes

Batch Processing:

- Batch node lookups into single BadgerDB transaction
- Parallelize independent subqueries
- Prefetch nodes during edge traversal

Acknowledgments

We thank the Lux Network community for feedback and testing, the BadgerDB team for their excellent key-value store, and the GraphQL Foundation for standardizing the query language.

References

- [1] Facebook, “GraphQL Specification,” *graphql.org*, 2015.
- [2] Dgraph Labs, “BadgerDB: Fast Key-Value Store in Go,” *GitHub*, 2017.
- [3] J. Kuszmaul, “Verkle Trees,” *Ethereum Research*, 2021.
- [4] Y. Yaniv et al., “The Graph: A Decentralized Query Protocol,” *White Paper*, 2018.
- [5] Team Rocket et al., “Avalanche: A Novel Metastable Consensus Protocol,” 2020.
- [6] Lux Industries, Inc., “Pulsar: Lattice-Based Threshold Signatures,” *LP-99*, 2025.