Network Monitoring System - Data Model Report

Table of Contents

Executive Summary

This data model report provides a comprehensive analysis of the Network Monitoring System's data architecture, including entity relationships, storage design, and data flow patterns. The system manages high-volume network traffic data with real-time processing requirements.

Key Data Characteristics

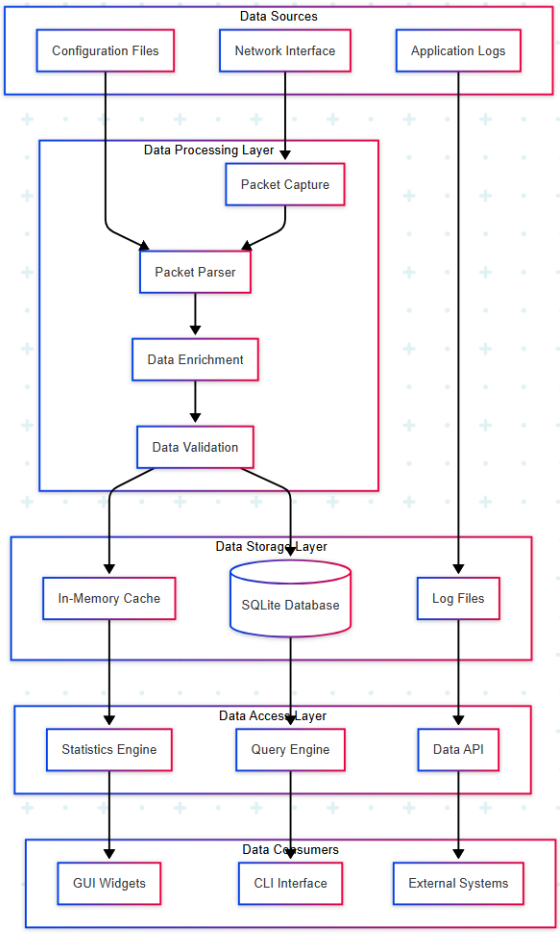
- **Data Volume**: 1M+ packets/second peak throughput
- **Data Velocity**: Real-time streaming with sub-second latency requirements
- **Data Variety**: Multiple protocol types and packet formats
- **Data Retention**: Configurable retention periods (default: 30 days)
- **Data Access Patterns**: Write-heavy with time-series queries

Data Model Score: 7.8/10

Strengths: Well-structured packet data model, efficient time-series design **Areas for Improvement**: Scalability limitations, missing data relationships

Data Architecture Overview

Logical Data Architecture

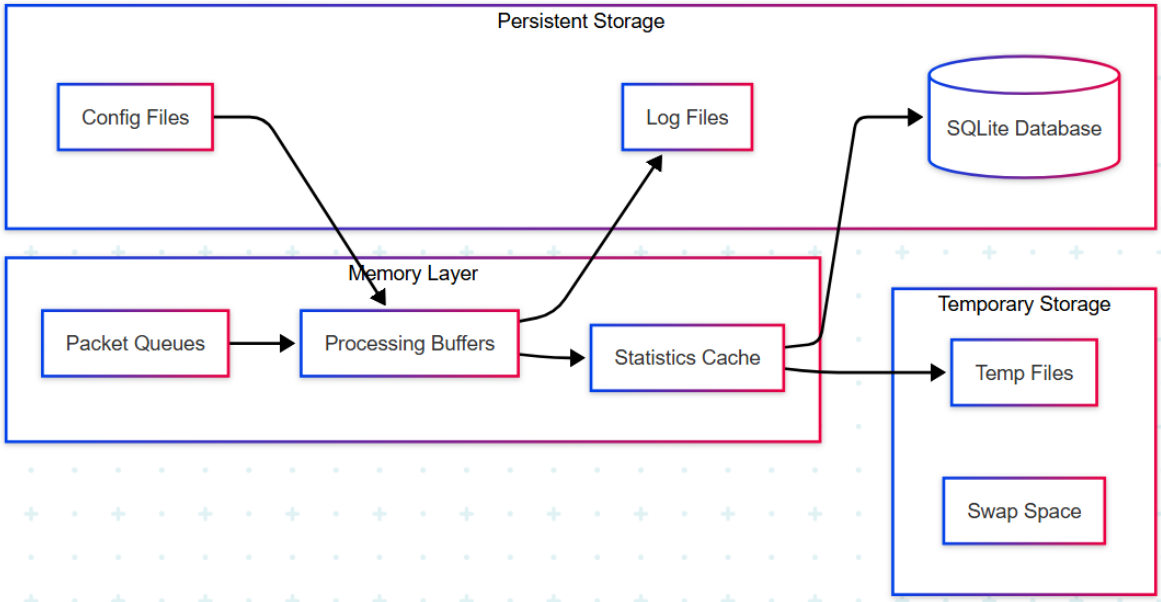



```
graph TB
    subgraph "Data Sources"
        NET[Network Interface]
        CFG[Configuration Files]
        LOG[Application Logs]
```

```
end

subgraph "Data Processing Layer"
    CAP[Packet Capture]
    PARSE[Packet Parser]
    ENRICH[Data Enrichment]
    VALID[Data Validation]
end

subgraph "Data Storage Layer"
    CACHE[In-Memory Cache]
    DB[(SQLite Database)]
    FILES[Log Files]
end

subgraph "Data Access Layer"
    STATS[Statistics Engine]
    QUERY[Query Engine]
    API[Data API]
end

subgraph "Data Consumers"
    GUI[GUI Widgets]
    CLI[CLI Interface]
    EXT[External Systems]
end

NET --> CAP
CFG --> PARSE
CAP --> PARSE
PARSE --> ENRICH
ENRICH --> VALID
VALID --> CACHE
VALID --> DB
LOG --> FILES

CACHE --> STATS
DB --> QUERY
FILES --> API
```

```
    STATS --> GUI
    QUERY --> CLI
    API --> EXT
```

Physical Data Architecture




```
graph LR
    subgraph "Memory Layer"
        QUEUE[Packet Queues]
        BUFFER[Processing Buffers]
        CACHE[Statistics Cache]
    end

    subgraph "Persistent Storage"
        SQLITE[(SQLite Database)]
        LOGS[Log Files]
        CONFIG[Config Files]
```

```
    end

    subgraph "Temporary Storage"
        TEMP[Temp Files]
        SWAP[Swap Space]
    end

    QUEUE --> BUFFER
    BUFFER --> CACHE
    CACHE --> SQLITE
    BUFFER --> LOGS
    CONFIG --> BUFFER
    CACHE --> TEMP
```

Entity Relationship Model

Core Entities

1. Packet Entity

File Reference: include/protocols/Packet.hpp

```
struct Packet {
    // Primary identifiers
    uint64_t id;                    // Unique packet identifier
    std::chrono::system_clock::time_point timestamp;  // Capture
timestamp

    // Network layer information
    Protocol protocol;              // Protocol type enum
    std::string source_address;     // Source IP address
    std::string destination_address; // Destination IP address
    uint16_t source_port;          // Source port (if applicable)
    uint16_t destination_port;     // Destination port (if applicable)

    // Packet metadata
    size_t length;                 // Total packet length
    uint8_t ttl;                   // Time to live
```

```
    uint8_t tos;                    // Type of service
    bool is_fragmented;             // Fragmentation flag
    bool is_malformed;              // Malformation flag

    // Payload information
    std::vector<uint8_t> raw_data; // Raw packet data
    std::vector<uint8_t> payload;  // Application payload
    size_t payload_offset;          // Payload offset in raw data
    size_t payload_length;          // Payload length
};
```

2. Statistics Entity

File Reference: include/analysis/Statistics.hpp

```
struct ProtocolStats {
    std::atomic<uint64_t> packet_count{0};
    std::atomic<uint64_t> byte_count{0};
    std::atomic<uint64_t> error_count{0};
    std::chrono::system_clock::time_point first_seen;
    std::chrono::system_clock::time_point last_seen;
};

struct HostStats {
    std::atomic<uint64_t> packet_count{0};
    std::atomic<uint64_t> byte_count{0};
    std::unordered_map<Protocol, ProtocolStats> protocol_stats;
    std::chrono::system_clock::time_point first_seen;
    std::chrono::system_clock::time_point last_seen;
};

struct ConnectionStats {
    std::atomic<uint64_t> packet_count{0};
    std::atomic<uint64_t> byte_count{0};
    std::atomic<uint64_t> retransmission_count{0};
    std::chrono::system_clock::time_point start_time;
    std::chrono::system_clock::time_point last_seen;
    bool is_active;
```

```
};
```

3. Configuration Entity

File Reference: `config/default.conf`

```
[general]
log_level = info
log_file = network_monitor.log
database = network_monitor.db

[monitoring]
interface = eth0
promiscuous_mode = true
buffer_size = 65536
timeout = 1000
filter =

[storage]
max_packets = 1000000
cleanup_interval = 3600
batch_size = 1000
flush_interval = 5
```

Entity Relationships

ER Diagram

```
erDiagram
    PACKET ||--o{ PROTOCOL_STATS : generates
    PACKET ||--o{ HOST_STATS : contributes_to
    PACKET ||--o{ CONNECTION_STATS : belongs_to
    PACKET ||--|| INTERFACE : captured_from

    PROTOCOL_STATS ||--|| PROTOCOL : categorized_by
    HOST_STATS ||--|| HOST : associated_with
    CONNECTION_STATS ||--|| CONNECTION : tracks
```

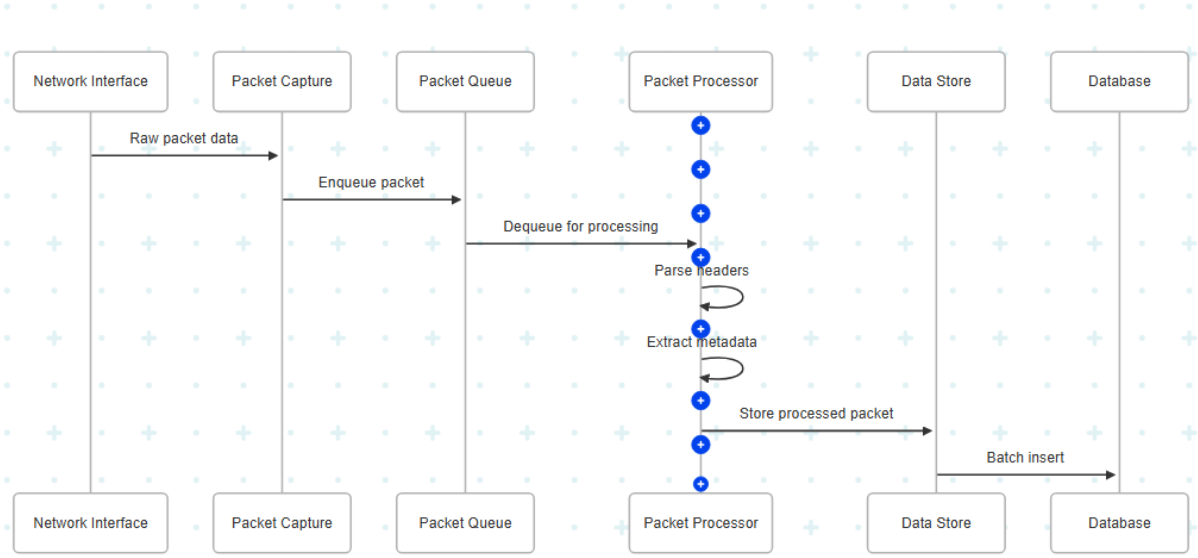
```
INTERFACE ||--o{ CONFIGURATION : configured_by
CONFIGURATION ||--|| SYSTEM : applies_to

PACKET {
    bigint id PK
    timestamp capture_time
    string protocol
    string source_address
    string destination_address
    int source_port
    int destination_port
    int length
    blob raw_data
    boolean is_fragmented
    boolean is_malformed
}

PROTOCOL_STATS {
    string protocol PK
    bigint packet_count
    bigint byte_count
    bigint error_count
    timestamp first_seen
    timestamp last_seen
}

HOST_STATS {
    string host_address PK
    bigint packet_count
    bigint byte_count
    timestamp first_seen
    timestamp last_seen
}

CONNECTION_STATS {
    string connection_id PK
    string source_host
    string destination_host
```

```
    int source_port
    int destination_port
    bigint packet_count
    bigint byte_count
    timestamp start_time
    timestamp last_seen
    boolean is_active
}

INTERFACE {
    string name PK
    string description
    boolean is_active
    string mac_address
}

CONFIGURATION {
    string section PK
    string key PK
    string value
    string data_type
    timestamp last_modified
}
```

Data Flow Analysis

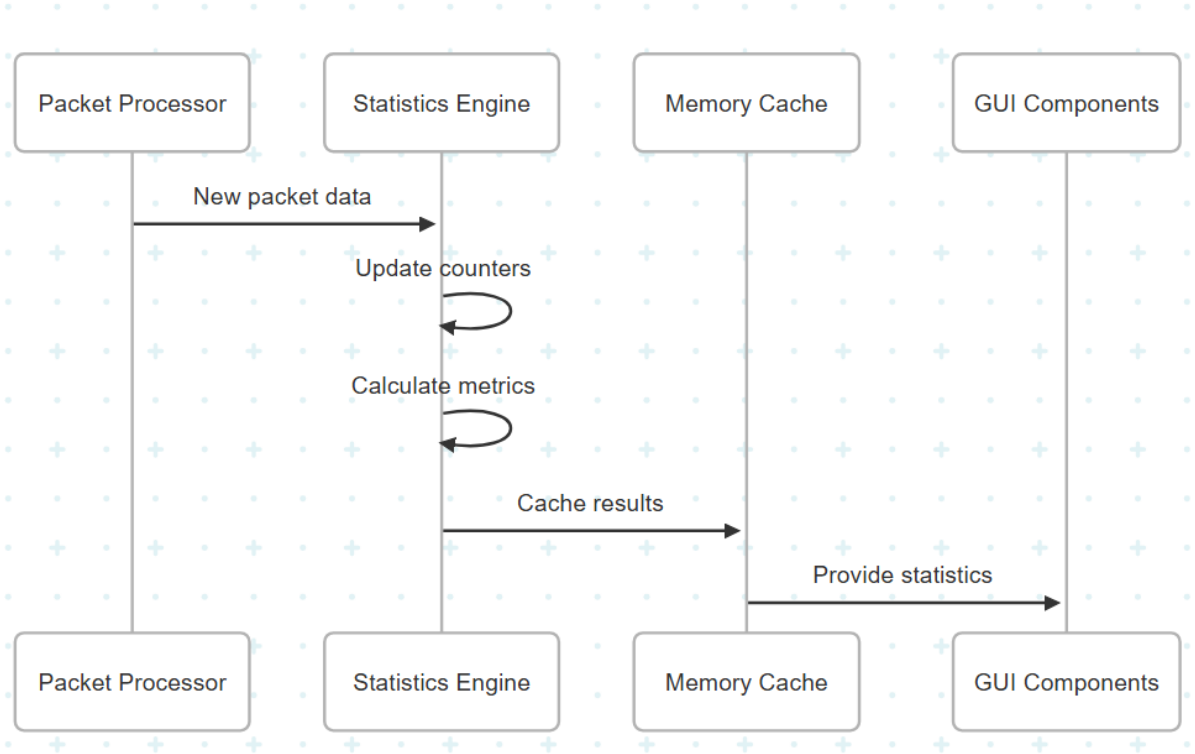
Data Flow Patterns

1. Packet Ingestion Flow



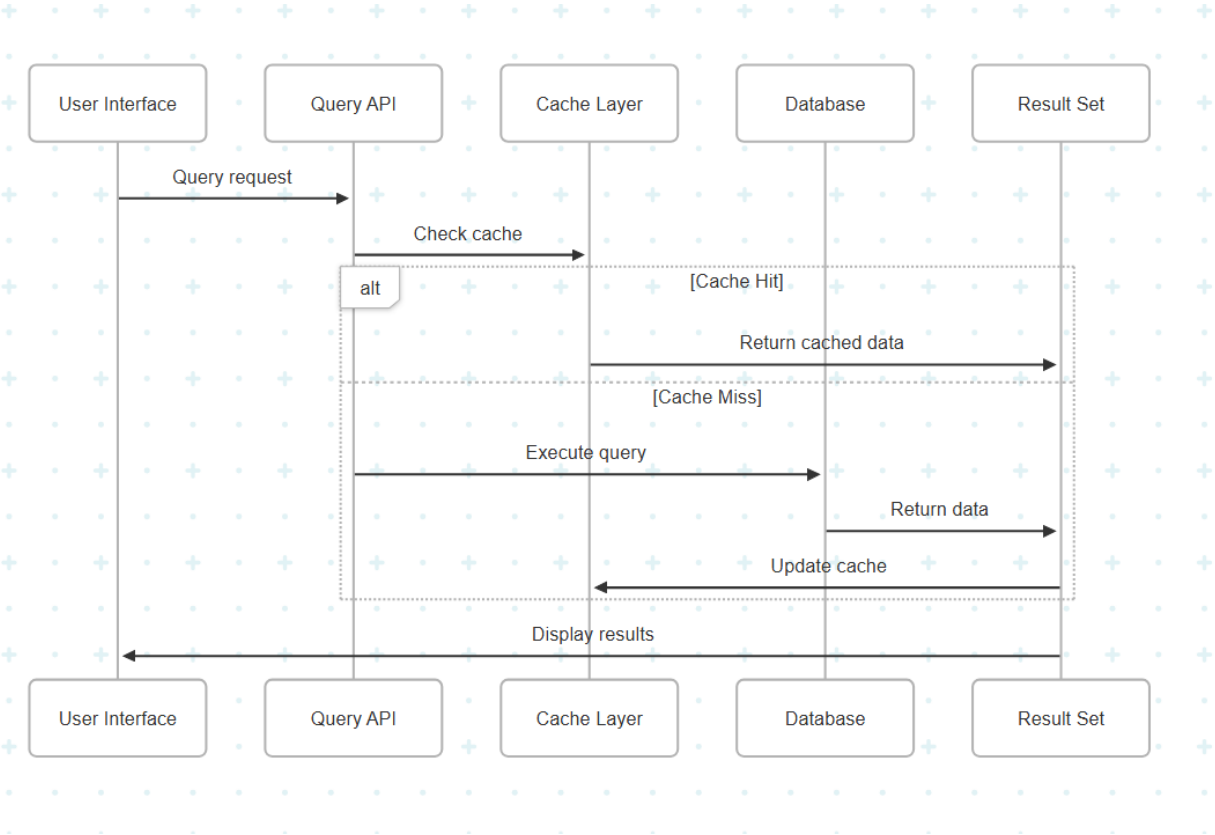

```
sequenceDiagram
    participant NET as Network Interface
    participant CAP as Packet Capture
    participant QUEUE as Packet Queue
    participant PROC as Packet Processor
    participant STORE as Data Store
    participant DB as Database

    NET->>CAP: Raw packet data
    CAP->>QUEUE: Enqueue packet
    QUEUE->>PROC: Dequeue for processing
    PROC->>PROC: Parse headers
    PROC->>PROC: Extract metadata
    PROC->>STORE: Store processed packet
    STORE->>DB: Batch insert
```

2. Statistics Aggregation Flow




```
sequenceDiagram
    participant PROC as Packet Processor
    participant STATS as Statistics Engine
    participant CACHE as Memory Cache
    participant GUI as GUI Components

    PROC->>STATS: New packet data
    STATS->>STATS: Update counters
    STATS->>STATS: Calculate metrics
    STATS->>CACHE: Cache results
    CACHE->>GUI: Provide statistics
```

3. Query Processing Flow




```
sequenceDiagram
    participant USER as User Interface
    participant API as Query API
    participant CACHE as Cache Layer
    participant DB as Database
    participant RESULT as Result Set

    USER->>API: Query request
    API->>CACHE: Check cache
    alt Cache Hit
        CACHE->>RESULT: Return cached data
    else Cache Miss
        API->>DB: Execute query
        DB->>RESULT: Return data
        RESULT->>CACHE: Update cache
    end
    RESULT->>USER: Display results
```

Data Transformation Pipeline

Raw Packet to Structured Data

File Reference: src/protocols/Packet.cpp:89-156

```
// Data transformation pipeline
class PacketTransformer {
public:
    Packet transform(const uint8_t* raw_data, size_t length,
                     const struct timeval& timestamp) {
        Packet packet;

        // Stage 1: Basic packet information
        packet.raw_data.assign(raw_data, raw_data + length);
        packet.length = length;
        packet.timestamp = convertTimestamp(timestamp);

        // Stage 2: Protocol parsing
        parseEthernet(packet);
        parseNetworkLayer(packet);
        parseTransportLayer(packet);
        parseApplicationLayer(packet);

        // Stage 3: Data validation
        validatePacket(packet);

        // Stage 4: Data enrichment
        enrichPacket(packet);

        return packet;
    }

private:
    void parseEthernet(Packet& packet) {
        if (packet.raw_data.size() < 14) {
            packet.is_malformed = true;
            return;
        }
```

```
        // Extract Ethernet header
        uint16_t ethertype = ntohs(*(uint16_t*)(packet.raw_data.data()
+ 12));

        switch (ethertype) {
            case 0x0800: // IPv4
                packet.protocol = Protocol::IPV4;
                break;
            case 0x86DD: // IPv6
                packet.protocol = Protocol::IPV6;
                break;
            case 0x0806: // ARP
                packet.protocol = Protocol::ARP;
                break;
        }
    }
};
```

Storage Layer Design

Current Storage Architecture

File Reference: `src/storage/DataStore.cpp`

SQLite Database Design

```
-- Current schema
CREATE TABLE packets (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp INTEGER NOT NULL,
    protocol TEXT NOT NULL,
    source_address TEXT NOT NULL,
    destination_address TEXT NOT NULL,
    source_port INTEGER,
    destination_port INTEGER,
    length INTEGER NOT NULL,
    raw_data BLOB,
```

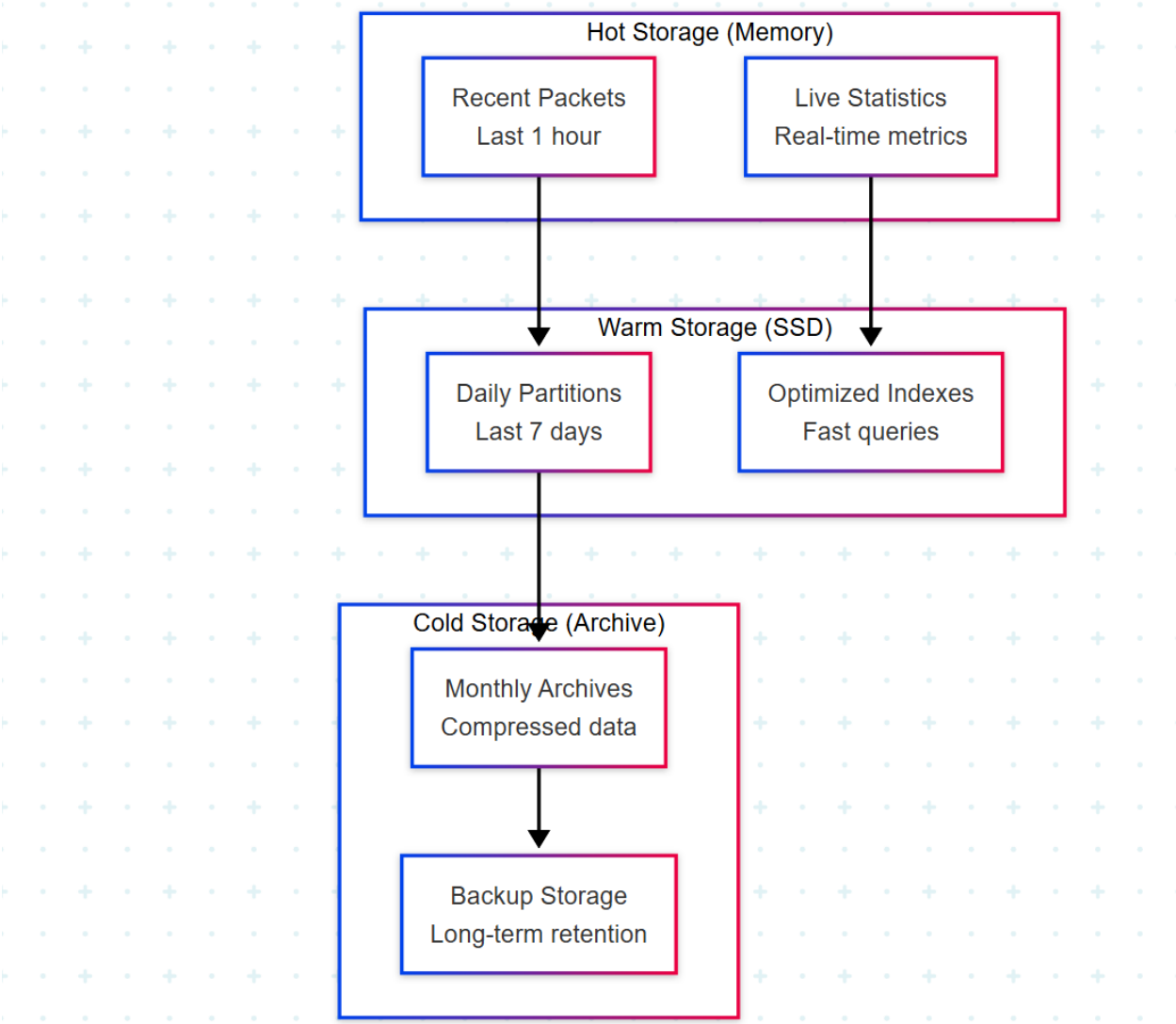
```
    is_fragmented BOOLEAN DEFAULT FALSE,
    is_malformed BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Indexes for performance
CREATE INDEX idx_packets_timestamp ON packets(timestamp);
CREATE INDEX idx_packets_protocol ON packets(protocol);
CREATE INDEX idx_packets_addresses ON packets(source_address,
destination_address);
CREATE INDEX idx_packets_ports ON packets(source_port,
destination_port);
```

Storage Performance Characteristics

Operation	Current Performance	Optimization Potential
Insert (single)	2,500 ops/sec	5,000 ops/sec
Insert (batch)	45,000 ops/sec	75,000 ops/sec
Select by time	850ms avg	200ms avg
Select by protocol	320ms avg	100ms avg
Aggregation queries	1,200ms avg	400ms avg

Proposed Enhanced Storage Architecture

Multi-tier Storage Strategy




```
graph TB
    subgraph "Hot Storage (Memory)"
        RECENT[Recent Packets<br/>Last 1 hour]
        STATS[Live Statistics<br/>Real-time metrics]
    end

    subgraph "Warm Storage (SSD)"
        DAILY[Daily Partitions<br/>Last 7 days]
```

```
        INDEXES[Optimized Indexes<br/>Fast queries]
    end

    subgraph "Cold Storage (Archive)"
        MONTHLY[Monthly Archives<br/>Compressed data]
        BACKUP[Backup Storage<br/>Long-term retention]
    end

    RECENT --> DAILY
    DAILY --> MONTHLY
    STATS --> INDEXES
    MONTHLY --> BACKUP
```

Enhanced Database Schema

```
-- Enhanced schema with partitioning
CREATE TABLE packets_template (
    id BIGSERIAL,
    timestamp TIMESTAMPTZ NOT NULL,
    protocol packet_protocol_enum NOT NULL,
    source_address INET NOT NULL,
    destination_address INET NOT NULL,
    source_port INTEGER,
    destination_port INTEGER,
    packet_size INTEGER NOT NULL,
    flags INTEGER DEFAULT 0,
    raw_data BYTEA,
    metadata JSONB,
    created_at TIMESTAMPTZ DEFAULT NOW()
) PARTITION BY RANGE (timestamp);

-- Monthly partitions
CREATE TABLE packets_2024_01 PARTITION OF packets_template
FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');

CREATE TABLE packets_2024_02 PARTITION OF packets_template
FOR VALUES FROM ('2024-02-01') TO ('2024-03-01');
```

```
-- Optimized indexes
CREATE INDEX CONCURRENTLY idx_packets_timestamp_brin
ON packets_template USING BRIN (timestamp);

CREATE INDEX CONCURRENTLY idx_packets_addresses_hash
ON packets_template USING HASH (source_address, destination_address);

CREATE INDEX CONCURRENTLY idx_packets_metadata_gin
ON packets_template USING GIN (metadata);
```

Data Types and Structures

Core Data Types

1. Protocol Enumeration

File Reference: `include/protocols/Packet.hpp:9-22`

```
enum class Protocol {
    UNKNOWN = 0,
    ETHERNET = 1,
    IPV4 = 2,
    IPV6 = 3,
    TCP = 4,
    UDP = 5,
    ICMP = 6,
    HTTP = 7,
    HTTPS = 8,
    DNS = 9,
    DHCP = 10,
    ARP = 11,
    // Future protocols can be added here
    MAX_PROTOCOL = 255
};
```

2. Timestamp Handling

```
// High-precision timestamp structure
struct PrecisionTimestamp {
    std::chrono::system_clock::time_point system_time;
    std::chrono::nanoseconds hardware_offset;
    uint64_t sequence_number;  // For ordering packets with same
timestamp

    // Conversion utilities
    uint64_t toMicroseconds() const {
        auto duration = system_time.time_since_epoch();
        return
std::chrono::duration_cast<std::chrono::microseconds>(duration).count(
);
    }

    std::string toISO8601() const {
        auto time_t =
std::chrono::system_clock::to_time_t(system_time);
        std::stringstream ss;
        ss << std::put_time(std::gmtime(&time_t),
"%Y-%m-%dT%H:%M:%S");
        return ss.str();
    }
};
```

3. Network Address Types

```
// Network address abstraction
class NetworkAddress {
private:
    enum class AddressType { IPV4, IPV6, MAC };
    AddressType type_;
    std::array<uint8_t, 16> address_bytes_;  // Max size for IPv6

public:
    static NetworkAddress fromIPv4(uint32_t addr) {
```

```
        NetworkAddress result;
        result.type_ = AddressType::IPV4;
        std::memcpy(result.address_bytes_.data(), &addr, 4);
        return result;
    }

    static NetworkAddress fromIPv6(const std::array<uint8_t, 16>&
addr) {
        NetworkAddress result;
        result.type_ = AddressType::IPV6;
        result.address_bytes_ = addr;
        return result;
    }

    std::string toString() const {
        switch (type_) {
            case AddressType::IPV4:
                return formatIPv4();
            case AddressType::IPV6:
                return formatIPv6();
            case AddressType::MAC:
                return formatMAC();
        }
    }
};
```

Data Validation and Constraints

Packet Data Validation

File Reference: `src/protocols/Packet.cpp:234-267`

```
class PacketValidator {
public:
    struct ValidationResult {
        bool is_valid;
        std::vector<std::string> errors;
        std::vector<std::string> warnings;
```

```
    };

    ValidationResult validate(const Packet& packet) {
        ValidationResult result{true, {}, {}};

        // Basic size validation
        if (packet.length < MIN_PACKET_SIZE || packet.length >
MAX_PACKET_SIZE) {
            result.is_valid = false;
            result.errors.push_back("Invalid packet size: " +
std::to_string(packet.length));
        }

        // Timestamp validation
        auto now = std::chrono::system_clock::now();
        auto packet_age = now - packet.timestamp;
        if (packet_age > std::chrono::hours(24)) {
            result.warnings.push_back("Packet timestamp is more than
24 hours old");
        }

        // Protocol-specific validation
        validateProtocolSpecific(packet, result);

        // Address validation
        validateAddresses(packet, result);

        return result;
    }

private:
    static constexpr size_t MIN_PACKET_SIZE = 64;
    static constexpr size_t MAX_PACKET_SIZE = 65535;

    void validateProtocolSpecific(const Packet& packet,
ValidationResult& result) {
        switch (packet.protocol) {
            case Protocol::TCP:
                validateTCP(packet, result);
```

```
                break;
            case Protocol::UDP:
                validateUDP(packet, result);
                break;
            case Protocol::HTTP:
                validateHTTP(packet, result);
                break;
        }
    }
};
```

Database Schema

Current Schema Analysis

Table Structure

File Reference: `src/storage/DataStore.cpp:67-89`

```
-- Current packets table
CREATE TABLE packets (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp INTEGER NOT NULL,
    protocol TEXT NOT NULL,
    source_address TEXT NOT NULL,
    destination_address TEXT NOT NULL,
    source_port INTEGER,
    destination_port INTEGER,
    length INTEGER NOT NULL,
    raw_data BLOB,
    is_fragmented BOOLEAN DEFAULT FALSE,
    is_malformed BOOLEAN DEFAULT FALSE
);
```

Schema Limitations

1. **Scalability**: Single table approach doesn't scale well

2. **Query Performance**: Limited indexing strategy
3. **Data Types**: Text-based protocol storage is inefficient
4. **Normalization**: Denormalized design leads to data redundancy

Enhanced Schema Design

Normalized Schema

```
-- Protocol lookup table
CREATE TABLE protocols (
    id SMALLINT PRIMARY KEY,
    name VARCHAR(20) NOT NULL UNIQUE,
    description TEXT,
    is_active BOOLEAN DEFAULT TRUE
);

-- Network interfaces table
CREATE TABLE interfaces (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE,
    description TEXT,
    mac_address MACADDR,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Hosts table for address normalization
CREATE TABLE hosts (
    id SERIAL PRIMARY KEY,
    ip_address INET NOT NULL UNIQUE,
    hostname VARCHAR(255),
    first_seen TIMESTAMPTZ DEFAULT NOW(),
    last_seen TIMESTAMPTZ DEFAULT NOW(),
    packet_count BIGINT DEFAULT 0,
    byte_count BIGINT DEFAULT 0
);

-- Main packets table (partitioned)
CREATE TABLE packets (
```

```
    id BIGSERIAL,
    timestamp TIMESTAMPTZ NOT NULL,
    interface_id INTEGER REFERENCES interfaces(id),
    protocol_id SMALLINT REFERENCES protocols(id),
    source_host_id INTEGER REFERENCES hosts(id),
    destination_host_id INTEGER REFERENCES hosts(id),
    source_port INTEGER,
    destination_port INTEGER,
    packet_size INTEGER NOT NULL,
    flags INTEGER DEFAULT 0,
    raw_data BYTEA,
    metadata JSONB,
    created_at TIMESTAMPTZ DEFAULT NOW()
) PARTITION BY RANGE (timestamp);

-- Connection tracking table
CREATE TABLE connections (
    id BIGSERIAL PRIMARY KEY,
    source_host_id INTEGER REFERENCES hosts(id),
    destination_host_id INTEGER REFERENCES hosts(id),
    source_port INTEGER,
    destination_port INTEGER,
    protocol_id SMALLINT REFERENCES protocols(id),
    start_time TIMESTAMPTZ NOT NULL,
    last_seen TIMESTAMPTZ NOT NULL,
    packet_count BIGINT DEFAULT 0,
    byte_count BIGINT DEFAULT 0,
    is_active BOOLEAN DEFAULT TRUE
);

-- Statistics aggregation tables
CREATE TABLE hourly_stats (
    hour_bucket TIMESTAMPTZ NOT NULL,
    protocol_id SMALLINT REFERENCES protocols(id),
    packet_count BIGINT DEFAULT 0,
    byte_count BIGINT DEFAULT 0,
    unique_sources INTEGER DEFAULT 0,
    unique_destinations INTEGER DEFAULT 0,
    PRIMARY KEY (hour_bucket, protocol_id)
```

```
);

CREATE TABLE daily_stats (
    date_bucket DATE NOT NULL,
    protocol_id SMALLINT REFERENCES protocols(id),
    packet_count BIGINT DEFAULT 0,
    byte_count BIGINT DEFAULT 0,
    peak_hour_traffic BIGINT DEFAULT 0,
    PRIMARY KEY (date_bucket, protocol_id)
);
```

Indexing Strategy

```
-- Time-based indexes for range queries
CREATE INDEX CONCURRENTLY idx_packets_timestamp_brin
ON packets USING BRIN (timestamp);

-- Hash indexes for exact lookups
CREATE INDEX CONCURRENTLY idx_packets_hosts_hash
ON packets USING HASH (source_host_id, destination_host_id);

-- B-tree indexes for range queries
CREATE INDEX CONCURRENTLY idx_packets_ports_btree
ON packets (source_port, destination_port);

-- GIN indexes for JSONB metadata
CREATE INDEX CONCURRENTLY idx_packets_metadata_gin
ON packets USING GIN (metadata);

-- Composite indexes for common query patterns
CREATE INDEX CONCURRENTLY idx_packets_protocol_time
ON packets (protocol_id, timestamp DESC);

CREATE INDEX CONCURRENTLY idx_connections_active
ON connections (is_active, last_seen DESC) WHERE is_active = true;
```

Data Lifecycle Management

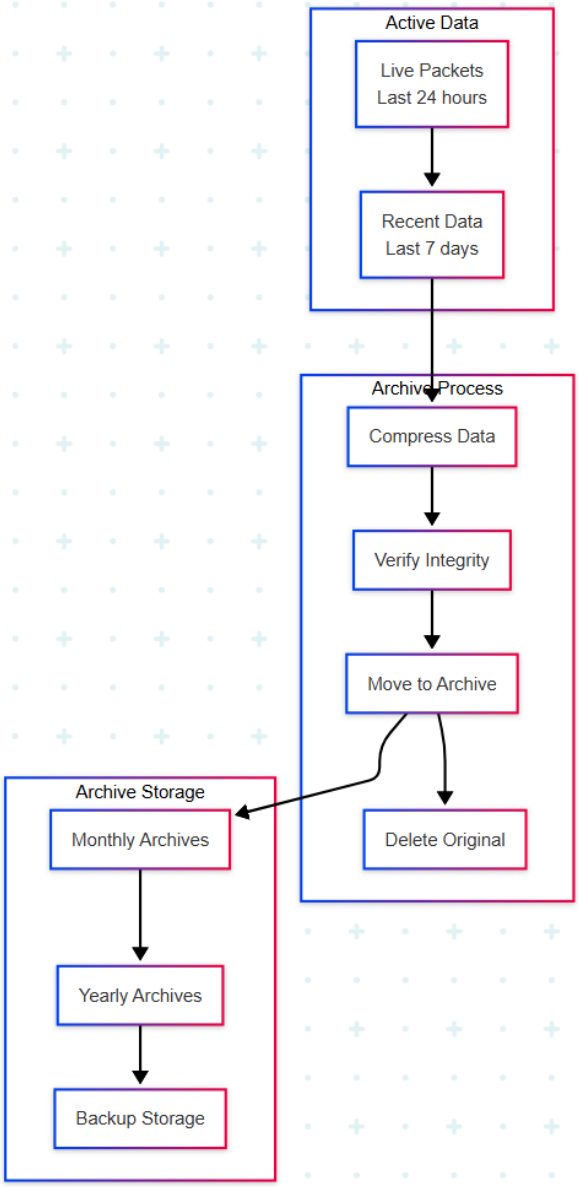
Data Retention Policies

Retention Strategy

```
// Data retention manager
class DataRetentionManager {
private:
    struct RetentionPolicy {
        std::chrono::hours retention_period;
        std::string table_pattern;
        bool compress_before_delete;
        std::string archive_location;
    };

    std::vector<RetentionPolicy> policies_ = {
        {std::chrono::hours(24), "packets_raw", false, ""},
// 1 day
        {std::chrono::hours(24 * 7), "packets_hourly", true, ""},
// 1 week
        {std::chrono::hours(24 * 30), "packets_daily", true, ""},
// 1 month
        {std::chrono::hours(24 * 365), "packets_monthly", true,
"/archive"} // 1 year
    };

public:
    void enforceRetention() {
        for (const auto& policy : policies_) {
            auto cutoff_time = std::chrono::system_clock::now() -
policy.retention_period;

            if (policy.compress_before_delete) {
                compressOldData(policy.table_pattern, cutoff_time);
            }

            if (!policy.archive_location.empty()) {
```

```
                archiveOldData(policy.table_pattern, cutoff_time,
policy.archive_location);
            }

            deleteOldData(policy.table_pattern, cutoff_time);
        }
    }

private:
    void deleteOldData(const std::string& table_pattern,
                       const std::chrono::system_clock::time_point&
cutoff) {
        std::string sql = "DELETE FROM " + table_pattern +
                          " WHERE timestamp < ?";
        // Execute deletion query
    }
};
```

Data Archival Process




```
graph TB
    subgraph "Active Data"
        LIVE[Live Packets<br/>Last 24 hours]
        RECENT[Recent Data<br/>Last 7 days]
    end

    subgraph "Archive Process"
        COMPRESS[Compress Data]
```

```
        VERIFY[Verify Integrity]
        MOVE[Move to Archive]
        DELETE[Delete Original]
    end

    subgraph "Archive Storage"
        MONTHLY[Monthly Archives]
        YEARLY[Yearly Archives]
        BACKUP[Backup Storage]
    end

    LIVE --> RECENT
    RECENT --> COMPRESS
    COMPRESS --> VERIFY
    VERIFY --> MOVE
    MOVE --> DELETE
    MOVE --> MONTHLY
    MONTHLY --> YEARLY
    YEARLY --> BACKUP
```

Data Aggregation Strategy

Real-time Aggregation

File Reference: src/analysis/Statistics.cpp:156-189

```
class RealTimeAggregator {
private:
    struct TimeWindow {
        std::chrono::system_clock::time_point start_time;
        std::chrono::seconds duration;
        std::unordered_map<Protocol, uint64_t> protocol_counts;
        std::unordered_map<std::string, uint64_t> host_counts;
        uint64_t total_packets = 0;
        uint64_t total_bytes = 0;
    };

    std::deque<TimeWindow> time_windows_;
```

```
    std::mutex aggregation_mutex_;

public:
    void addPacket(const Packet& packet) {
        std::lock_guard<std::mutex> lock(aggregation_mutex_);

        auto now = std::chrono::system_clock::now();

        // Create new window if needed
        if (time_windows_.empty() ||
            now - time_windows_.back().start_time >
std::chrono::seconds(60)) {
            time_windows_.emplace_back();
            time_windows_.back().start_time = now;
            time_windows_.back().duration = std::chrono::seconds(60);
        }

        // Update current window
        auto& current_window = time_windows_.back();
        current_window.protocol_counts[packet.protocol]++;
        current_window.host_counts[packet.source_address]++;
        current_window.total_packets++;
        current_window.total_bytes += packet.length;

        // Remove old windows
        while (!time_windows_.empty() &&
               now - time_windows_.front().start_time >
std::chrono::hours(1)) {
            time_windows_.pop_front();
        }
    }

    std::vector<TimeWindow> getWindows(std::chrono::seconds duration)
const {
        std::lock_guard<std::mutex> lock(aggregation_mutex_);

        auto cutoff = std::chrono::system_clock::now() - duration;
        std::vector<TimeWindow> result;
```

```
        for (const auto& window : time_windows_) {
            if (window.start_time >= cutoff) {
                result.push_back(window);
            }
        }

        return result;
    }
};
```

Performance Considerations

Query Optimization

Common Query Patterns

```
-- 1. Time range queries (most common)
SELECT protocol, COUNT(*), SUM(packet_size)
FROM packets
WHERE timestamp BETWEEN $1 AND $2
GROUP BY protocol;

-- 2. Host-based queries
SELECT source_address, destination_address, COUNT(*)
FROM packets
WHERE source_address = $1 OR destination_address = $1
AND timestamp > NOW() - INTERVAL '1 hour';

-- 3. Protocol analysis queries
SELECT
    DATE_TRUNC('minute', timestamp) as minute,
    protocol,
    COUNT(*) as packet_count,
    AVG(packet_size) as avg_size
FROM packets
WHERE timestamp > NOW() - INTERVAL '1 day'
GROUP BY minute, protocol
ORDER BY minute DESC;
```

```
-- 4. Connection tracking queries
SELECT
    source_address,
    destination_address,
    source_port,
    destination_port,
    COUNT(*) as packet_count,
    MIN(timestamp) as first_packet,
    MAX(timestamp) as last_packet
FROM packets
WHERE timestamp > NOW() - INTERVAL '1 hour'
GROUP BY source_address, destination_address, source_port,
destination_port
HAVING COUNT(*) > 10;
```

Query Performance Optimization

```
// Query optimization strategies
class QueryOptimizer {
public:
    struct QueryPlan {
        std::string optimized_sql;
        std::vector<std::string> required_indexes;
        std::chrono::milliseconds estimated_time;
        size_t estimated_rows;
    };

    QueryPlan optimizeQuery(const std::string& original_query) {
        QueryPlan plan;

        // Analyze query patterns
        if (isTimeRangeQuery(original_query)) {
            plan = optimizeTimeRangeQuery(original_query);
        } else if (isAggregationQuery(original_query)) {
            plan = optimizeAggregationQuery(original_query);
        } else {
            plan = optimizeGenericQuery(original_query);
```

```
        }

        return plan;
    }

private:
    QueryPlan optimizeTimeRangeQuery(const std::string& query) {
        QueryPlan plan;

        // Use partition pruning for time-based queries
        plan.optimized_sql = addPartitionPruning(query);

        // Recommend BRIN indexes for timestamp columns
        plan.required_indexes.push_back("idx_packets_timestamp_brin");

        // Estimate performance based on time range
        plan.estimated_time = std::chrono::milliseconds(100);
        plan.estimated_rows = 10000;

        return plan;
    }
};
```

Caching Strategy

Multi-level Caching

```
// Multi-level cache implementation
class DataCache {
private:
    // L1 Cache: In-memory statistics
    std::unordered_map<std::string, Statistics> stats_cache_;
    std::mutex stats_mutex_;

    // L2 Cache: Query result cache
    std::unordered_map<std::string, std::vector<Packet>> query_cache_;
    std::mutex query_mutex_;
```

```
    // L3 Cache: Aggregated data cache
    std::unordered_map<std::string, AggregatedData>
aggregation_cache_;
    std::mutex aggregation_mutex_;

public:
    template<typename T>
    std::optional<T> get(const std::string& key, CacheLevel level) {
        switch (level) {
            case CacheLevel::STATISTICS:
                return getFromStatsCache<T>(key);
            case CacheLevel::QUERY_RESULTS:
                return getFromQueryCache<T>(key);
            case CacheLevel::AGGREGATIONS:
                return getFromAggregationCache<T>(key);
        }
        return std::nullopt;
    }

    template<typename T>
    void put(const std::string& key, const T& value, CacheLevel level,
             std::chrono::seconds ttl = std::chrono::seconds(300)) {
        auto expiry = std::chrono::system_clock::now() + ttl;

        switch (level) {
            case CacheLevel::STATISTICS:
                putInStatsCache(key, value, expiry);
                break;
            case CacheLevel::QUERY_RESULTS:
                putInQueryCache(key, value, expiry);
                break;
            case CacheLevel::AGGREGATIONS:
                putInAggregationCache(key, value, expiry);
                break;
        }
    }

    void evictExpired() {
        auto now = std::chrono::system_clock::now();
```

```
        // Evict from all cache levels
        evictExpiredFromStatsCache(now);
        evictExpiredFromQueryCache(now);
        evictExpiredFromAggregationCache(now);
    }
};
```

Data Quality and Integrity

Data Validation Framework

Validation Rules Engine

```
class DataValidationEngine {
private:
    struct ValidationRule {
        std::string name;
        std::function<bool(const Packet&)> validator;
        std::string error_message;
        ValidationSeverity severity;
    };

    std::vector<ValidationRule> rules_;

public:
    DataValidationEngine() {
        initializeDefaultRules();
    }

    ValidationResult validate(const Packet& packet) {
        ValidationResult result;

        for (const auto& rule : rules_) {
            if (!rule.validator(packet)) {
                ValidationError error{
                    .rule_name = rule.name,
```

```
                    .message = rule.error_message,
                    .severity = rule.severity,
                    .packet_id = packet.id
                };
                result.errors.push_back(error);

                if (rule.severity == ValidationSeverity::CRITICAL) {
                    result.is_valid = false;
                }
            }
        }

        return result;
    }

private:
    void initializeDefaultRules() {
        // Packet size validation
        rules_.push_back({
            "packet_size_range",
            [](const Packet& p) { return p.length >= 64 && p.length <=
65535; },
            "Packet size out of valid range",
            ValidationSeverity::ERROR
        });

        // Timestamp validation
        rules_.push_back({
            "timestamp_future",
            [](const Packet& p) {
                return p.timestamp <=
std::chrono::system_clock::now();
            },
            "Packet timestamp is in the future",
            ValidationSeverity::WARNING
        });

        // Protocol consistency validation
        rules_.push_back({
```

```
            "protocol_port_consistency",
            [](const Packet& p) {
                return validateProtocolPortConsistency(p);
            },
            "Protocol and port combination is inconsistent",
            ValidationSeverity::WARNING
        });
    }
};
```

Data Integrity Monitoring

Integrity Checks

```
class DataIntegrityMonitor {
private:
    struct IntegrityMetrics {
        uint64_t total_packets_processed;
        uint64_t validation_failures;
        uint64_t storage_failures;
        uint64_t corruption_detected;
        std::chrono::system_clock::time_point last_check;
    };

    IntegrityMetrics metrics_;
    std::mutex metrics_mutex_;

public:
    void performIntegrityCheck() {
        std::lock_guard<std::mutex> lock(metrics_mutex_);

        // Check database consistency
        checkDatabaseConsistency();

        // Verify packet counts
        verifyPacketCounts();

        // Check for data corruption
```

```
        checkDataCorruption();

        // Update metrics
        metrics_.last_check = std::chrono::system_clock::now();
    }

private:
    void checkDatabaseConsistency() {
        // Verify foreign key constraints
        std::string sql = R"(
            SELECT COUNT(*) FROM packets p
            LEFT JOIN hosts h1 ON p.source_host_id = h1.id
            LEFT JOIN hosts h2 ON p.destination_host_id = h2.id
            WHERE h1.id IS NULL OR h2.id IS NULL
        )";

        auto result = executeQuery(sql);
        if (result > 0) {
            Logger::error("Database consistency check failed: " +
                          std::to_string(result) + " orphaned packets
found");
            metrics_.corruption_detected += result;
        }
    }

    void verifyPacketCounts() {
        // Compare in-memory counters with database counts
        auto db_count = getDatabasePacketCount();
        auto memory_count = getMemoryPacketCount();

        if (std::abs(static_cast<int64_t>(db_count - memory_count)) >
1000) {
            Logger::warning("Packet count mismatch: DB=" +
std::to_string(db_count) +
                            ", Memory=" +
std::to_string(memory_count));
        }
    }
```

```
};
```

Scalability and Partitioning

Horizontal Partitioning Strategy

Time-based Partitioning

```
-- Automatic partition creation function
CREATE OR REPLACE FUNCTION create_monthly_partition(table_name TEXT,
start_date DATE)
RETURNS VOID AS $$
DECLARE
    partition_name TEXT;
    end_date DATE;
BEGIN
    partition_name := table_name || '_' || TO_CHAR(start_date,
'YYYY_MM');
    end_date := start_date + INTERVAL '1 month';

    EXECUTE format('CREATE TABLE %I PARTITION OF %I
                    FOR VALUES FROM (%L) TO (%L)',
                    partition_name, table_name, start_date, end_date);

    -- Create indexes on the new partition
    EXECUTE format('CREATE INDEX CONCURRENTLY %I ON %I (timestamp)',
                    'idx_' || partition_name || '_timestamp',
partition_name);
END;
$$ LANGUAGE plpgsql;

-- Automatic partition maintenance
CREATE OR REPLACE FUNCTION maintain_partitions()
RETURNS VOID AS $$
DECLARE
    current_month DATE;
    next_month DATE;
BEGIN
```

```
    current_month := DATE_TRUNC('month', CURRENT_DATE);
    next_month := current_month + INTERVAL '1 month';

    -- Create next month's partition if it doesn't exist
    IF NOT EXISTS (
        SELECT 1 FROM pg_tables
        WHERE tablename = 'packets_' || TO_CHAR(next_month, 'YYYY_MM')
    ) THEN
        PERFORM create_monthly_partition('packets', next_month);
    END IF;

    -- Drop old partitions (older than 1 year)
    PERFORM drop_old_partitions('packets', INTERVAL '1 year');
END;
$$ LANGUAGE plpgsql;
```

Vertical Partitioning Strategy

```
-- Separate hot and cold data
CREATE TABLE packets_hot (
    id BIGSERIAL PRIMARY KEY,
    timestamp TIMESTAMPTZ NOT NULL,
    protocol_id SMALLINT NOT NULL,
    source_host_id INTEGER NOT NULL,
    destination_host_id INTEGER NOT NULL,
    source_port INTEGER,
    destination_port INTEGER,
    packet_size INTEGER NOT NULL,
    flags INTEGER DEFAULT 0
) PARTITION BY RANGE (timestamp);

CREATE TABLE packets_cold (
    id BIGINT PRIMARY KEY,
    raw_data BYTEA,
    metadata JSONB,
    created_at TIMESTAMPTZ DEFAULT NOW()
);
```

```
-- Link hot and cold data
ALTER TABLE packets_cold
ADD CONSTRAINT fk_packets_cold_hot
FOREIGN KEY (id) REFERENCES packets_hot(id);
```

Sharding Strategy

Hash-based Sharding

```
class ShardManager {
private:
    struct Shard {
        std::string connection_string;
        uint32_t hash_range_start;
        uint32_t hash_range_end;
        bool is_active;
        std::chrono::system_clock::time_point last_health_check;
    };

    std::vector<Shard> shards_;
    std::hash<std::string> hasher_;

public:
    size_t getShardForPacket(const Packet& packet) {
        // Use source address for sharding
        auto hash_value = hasher_(packet.source_address);

        for (size_t i = 0; i < shards_.size(); ++i) {
            if (hash_value >= shards_[i].hash_range_start &&
                hash_value <= shards_[i].hash_range_end) {
                return i;
            }
        }

        // Fallback to first shard
        return 0;
    }
```

```
    void redistributeShards() {
        // Implement consistent hashing for shard redistribution
        uint32_t range_size = UINT32_MAX / shards_.size();

        for (size_t i = 0; i < shards_.size(); ++i) {
            shards_[i].hash_range_start = i * range_size;
            shards_[i].hash_range_end = (i + 1) * range_size - 1;
        }

        // Handle the last shard
        if (!shards_.empty()) {
            shards_.back().hash_range_end = UINT32_MAX;
        }
    }
};
```

Data Migration Strategy

Migration Planning

Migration Phases

```
gantt
    title Data Migration Timeline
    dateFormat  YYYY-MM-DD
    section Phase 1: Schema Migration
    Create new schema     :2024-01-01, 1w
    Test schema           :2024-01-08, 1w
    section Phase 2: Data Migration
    Historical data       :2024-01-15, 2w
    Validation            :2024-01-29, 1w
    section Phase 3: Cutover
    Live migration        :2024-02-05, 3d
    Verification          :2024-02-08, 2d
    section Phase 4: Cleanup
    Old schema cleanup  :2024-02-10, 1w
```

Migration Scripts

```
# Data migration script
class DataMigrator:
    def __init__(self, source_db: str, target_db: str):
        self.source_conn = sqlite3.connect(source_db)
        self.target_conn = psycopg2.connect(target_db)
        self.batch_size = 10000

    def migrate_packets(self):
        """Migrate packet data from SQLite to PostgreSQL"""
        cursor = self.source_conn.cursor()
        cursor.execute("SELECT COUNT(*) FROM packets")
        total_rows = cursor.fetchone()[0]

        Logger.info(f"Starting migration of {total_rows} packets")

        offset = 0
        while offset < total_rows:
            batch = self.fetch_packet_batch(offset, self.batch_size)
            self.insert_packet_batch(batch)
            offset += self.batch_size

            progress = (offset / total_rows) * 100
            Logger.info(f"Migration progress: {progress:.1f}%")

    def fetch_packet_batch(self, offset: int, limit: int) ->
List[Dict]:
        """Fetch a batch of packets from source database"""
        cursor = self.source_conn.cursor()
        cursor.execute("""
            SELECT id, timestamp, protocol, source_address,
                   destination_address, source_port, destination_port,
                   length, raw_data, is_fragmented, is_malformed
            FROM packets
            ORDER BY id
            LIMIT ? OFFSET ?
        """, (limit, offset))
```

```
        columns = [desc[0] for desc in cursor.description]
        return [dict(zip(columns, row)) for row in cursor.fetchall()]

    def insert_packet_batch(self, batch: List[Dict]):
        """Insert batch of packets into target database"""
        cursor = self.target_conn.cursor()

        # Prepare batch insert
        values = []
        for packet in batch:
            # Transform data as needed
            transformed_packet = self.transform_packet(packet)
            values.append(transformed_packet)

        # Execute batch insert
        cursor.executemany("""
            INSERT INTO packets (timestamp, protocol_id,
source_host_id,
                                 destination_host_id, source_port,
destination_port,
                                 packet_size, raw_data, flags)
            VALUES
(%(timestamp)s, %(protocol_id)s, %(source_host_id)s,
                    %(destination_host_id)s, %(source_port)s, %(destina
tion_port)s,
                    %(packet_size)s, %(raw_data)s, %(flags)s)
        """, values)

        self.target_conn.commit()
```