DB25: Modern Query Processing Engine Architecture
A Comprehensive Implementation and Extension Framework

Chiradip Mandal
chiradip@github.com

DB25: C++17 PostgreSQL-Compatible Query Engine
Graduate Database Systems Course
Advanced Implementation Tutorial

August 19, 2025

Abstract

This paper presents a comprehensive analysis and implementation guide for a modern query processing engine built with C++17, featuring PostgreSQL-compatible SQL parsing, cost-based optimization, and vectorized execution. The implementation demonstrates the complete pipeline from SQL parsing through physical execution, serving both as an educational tool for graduate database systems courses and as a foundation for advanced database research. We detail the current architecture, identify extension points for production-ready enhancements, and provide a roadmap for implementing missing components including storage management, transaction processing, and advanced optimization techniques.

Keywords: Query Processing, Database Systems, Cost-Based Optimization, Vectorized Execution, C++17, PostgreSQL

Contents

1 Introduction

Modern database management systems represent some of the most complex software systems ever built, incorporating decades of research in query optimization, storage management, and concurrent processing. This paper presents a systematic approach to understanding and implementing a query processing engine that demonstrates core database concepts while providing a foundation for advanced research and development.

1.1 System Overview

Our implementation provides a complete query processing pipeline:

1. **SQL Parsing** using PostgreSQL's `libpg_query`
2. **Logical Planning** with cost-based optimization
3. **Physical Planning** with operator selection
4. **Vectorized Execution** with parallel processing
5. **Schema Management** with DDL support

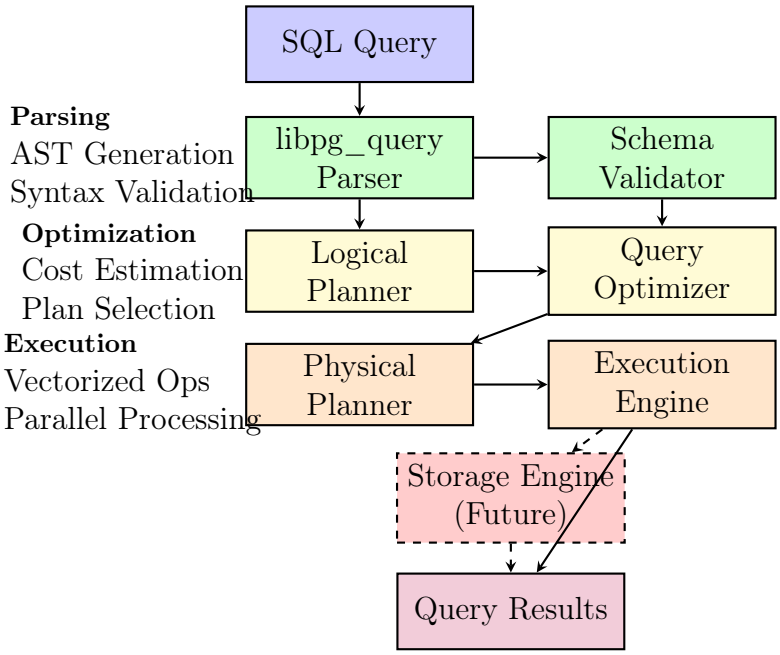


Figure 1: Query Processing Architecture Overview

1.2 Educational Objectives

This implementation serves multiple educational purposes:

- **Conceptual Understanding**: Demonstrates how SQL queries are transformed into executable plans

- **Implementation Details**: Shows practical considerations in building query processors
- **Performance Analysis**: Illustrates cost models and optimization techniques
- **Extension Framework**: Provides clear pathways for adding production features

1.3 Algorithmic Notation

Notation 1.1 (Algorithm Pseudocode Syntax)**.** *This document uses the* `algorithmic` *package syntax for all algorithm descriptions. The key elements are:*

- ***Control Structures**:*
  - `\FOR{condition} ... \ENDFOR` *- For loops*
  - `\WHILE{condition} ... \ENDWHILE` *- While loops*
  - `\IF{condition} ... \ENDIF` *- Conditional statements*
- ***Operations**:*
  - `\STATE statement` *- Single algorithmic step*
  - `\CALL{Function}{args}` *- Function calls*
  - `\COMMENT{text}` *- Explanatory comments*
- ***Mathematical Notation**:*
  - $\leftarrow$ *- Assignment operator*
  - $\neg$ *- Logical negation*
  - $\vee, \wedge$ *- Logical OR, AND*
  - $|T|$ *- Cardinality (size) of relation* T
- ***Specifications**:*
  - `\REQUIRE` *- Algorithm preconditions*
  - `\ENSURE` *- Algorithm postconditions*
  - `\RETURN` *- Algorithm return value*

This unified notation ensures consistency across all algorithmic descriptions and maintains academic standards for algorithm presentation.

2 System Architecture

2.1 Component Hierarchy

The system follows a layered architecture with clear separation of concerns:

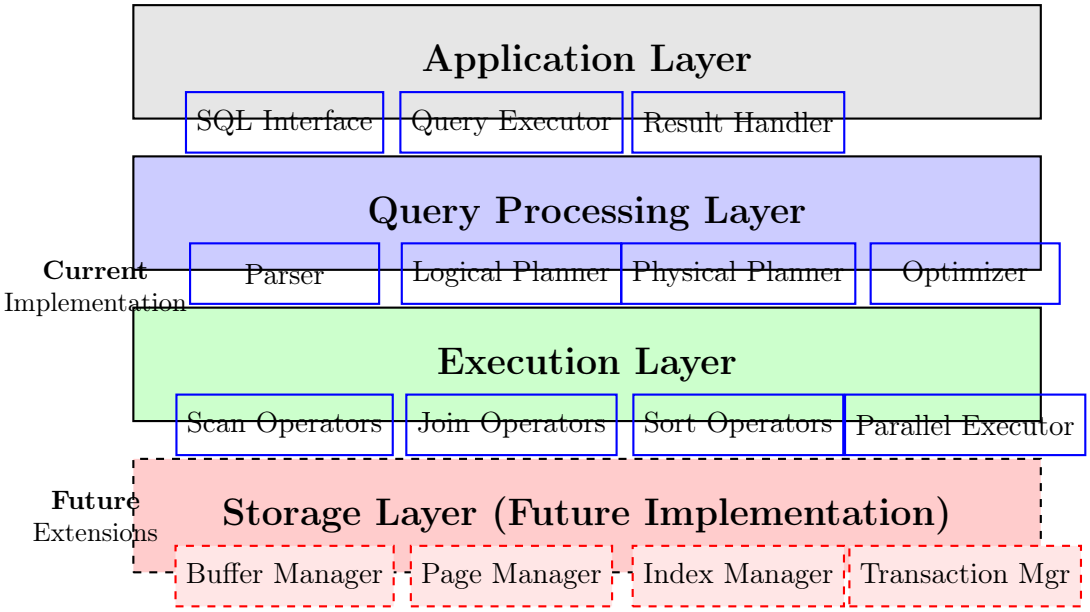


Figure 2: System Component Hierarchy

2.2 Core Classes and Interfaces

The implementation uses modern C++17 features and follows SOLID principles:

Listing 1: Core Interface Definitions

```
namespace pg {
    // Base query plan node
    struct LogicalPlanNode {
        PlanNodeType type;
        PlanCost cost;
        std::vector<LogicalPlanNodePtr> children;
        std::vector<std::string> output_columns;

        virtual std::string to_string(int indent = 0) const = 0;
        virtual LogicalPlanNodePtr copy() const = 0;
    };

    // Physical execution interface
    struct PhysicalPlanNode {
        PhysicalOperatorType type;
        ExecutionStats actual_stats;

        virtual TupleBatch get_next_batch() = 0;
        virtual void reset() = 0;
        virtual void initialize(ExecutionContext* ctx) = 0;
    };

    // Main planner interface
    class QueryPlanner {
        std::shared_ptr<DatabaseSchema> schema_;
        CostModel cost_model_;

```

```
    public:
        LogicalPlan create_plan(const std::string& query);
        std::vector<LogicalPlan> generate_alternatives(const std::
            string& query);
        void optimize_plan(LogicalPlan& plan);
    };
}
```

3 Query Parsing and Validation

3.1 Integration with libpg_query

We leverage PostgreSQL's proven parsing infrastructure through `libpg_query`, ensuring compatibility with PostgreSQL SQL syntax:

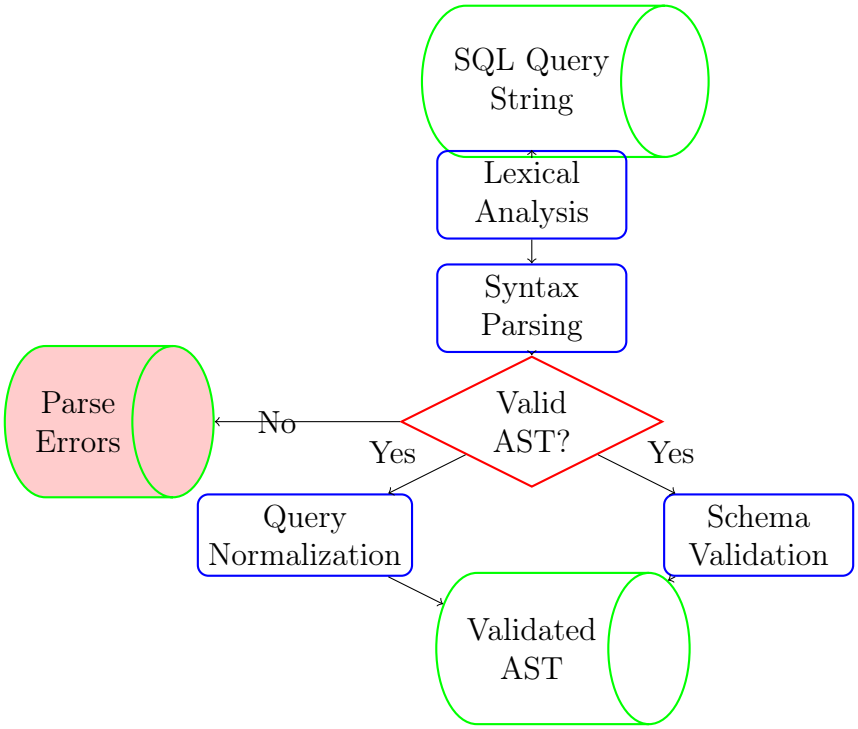


Figure 3: SQL Parsing Pipeline

3.2 AST Processing

The Abstract Syntax Tree (AST) processing involves multiple validation phases:

Listing 2: AST Processing Example

```
class PgQueryWrapper {
    ParseResult parse(const std::string& query) {
        ParseResult result;

        // Use libpg_query for parsing
        auto pg_result = pg_query_parse(query.c_str());
```

```
        if (pg_result.error) {
            result.is_valid = false;
            result.errors.push_back(pg_result.error->message);
        } else {
            result.is_valid = true;
            result.parse_tree = pg_result.parse_tree;

            // Extract query components
            extract_table_references(pg_result.parse_tree, result);
            extract_column_references(pg_result.parse_tree, result);
        }

        pg_query_free_parse_result(pg_result);
        return result;
    }
};
```

4 Logical Query Planning

4.1 Plan Node Types

The logical planning phase creates a tree of logical operators:

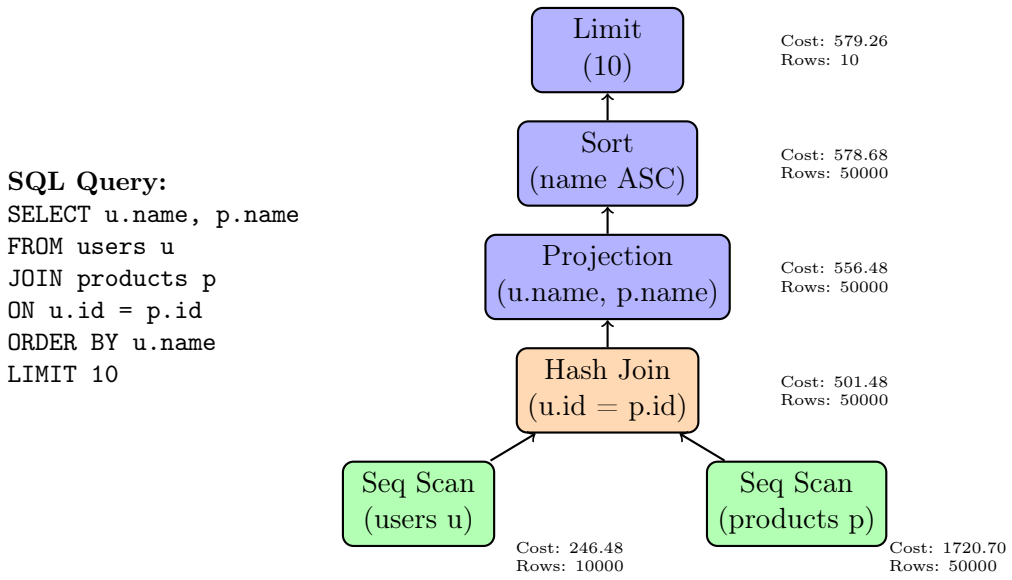


Figure 4: Logical Query Plan Tree Structure

4.2 Cost Model

The cost model estimates execution costs using statistical information:

Definition 4.1 (Cost Model). *For any logical plan node N, the total cost is computed as:*

$$C_{total}(N) = C_{startup}(N) + C_{run}(N) \tag{1}$$
$$C_{run}(N) = C_{cpu}(N) + C_{io}(N) \tag{2}$$
$$C_{cpu}(N) = rows(N) \times c_{cpu} \tag{3}$$
$$C_{io}(N) = pages(N) \times c_{io} \tag{4}$$

where c_{cpu} and c_{io} are cost coefficients.

Listing 3: Cost Calculation Implementation

```
struct PlanCost {
    double startup_cost = 0.0;
    double total_cost = 0.0;
    size_t estimated_rows = 0;
    double estimated_width = 0.0;

    // Cost calculation for sequential scan
    static PlanCost calculate_seq_scan_cost(const TableStats&
       stats) {
        PlanCost cost;
        cost.startup_cost = 0.0;

        // IO cost: pages * seq_page_cost
        double io_cost = stats.pages * SEQ_PAGE_COST;

        // CPU cost: tuples * cpu_tuple_cost
        double cpu_cost = stats.row_count * CPU_TUPLE_COST;

        cost.total_cost = cost.startup_cost + io_cost + cpu_cost;
        cost.estimated_rows = stats.row_count;
        cost.estimated_width = stats.avg_row_size;

        return cost;
    }
};
```

4.3 Query Optimization Rules

The optimizer applies transformation rules to improve query plans:

5 Physical Query Planning

5.1 Operator Selection

Physical planning converts logical operators into executable physical operators:

5.2 Memory Management

The execution engine implements sophisticated memory management:

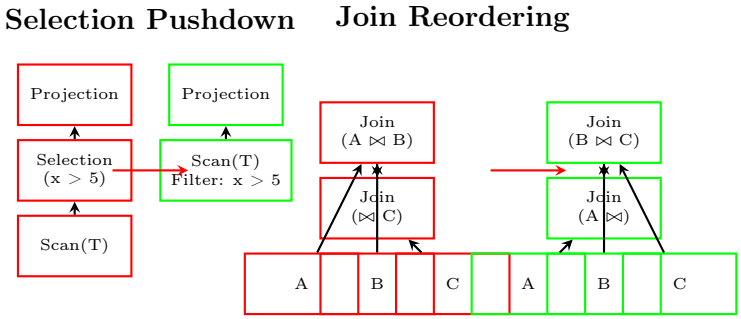


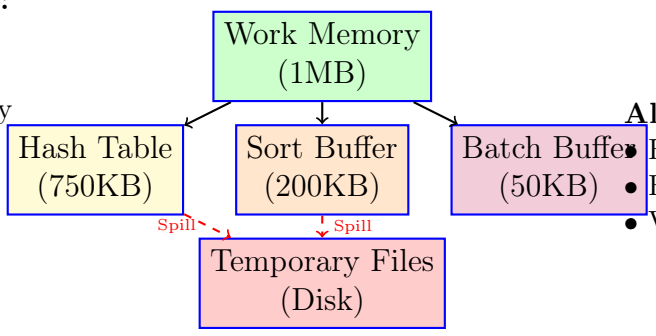
Figure 5: Query Optimization Transformations

Table 1: Logical to Physical Operator Mapping

Logical Operator	**Physical Options**	**Selection Criteria**
Table Scan	Sequential Scan	Default choice
	Index Scan	Selective predicates
	Parallel Seq Scan	Large tables
Join	Nested Loop Join	Small tables
	Hash Join	One small, one large table
	Sort-Merge Join	Both inputs sorted
Aggregation	Hash Aggregate	GROUP BY queries
	Sort Aggregate	Sorted input
Sort	In-Memory Sort	Small datasets
	External Sort	Large datasets

Memory Hierarchy:

- Work memory limit
- Operator budgets
- Spill-to-disk strategy



Algorithms:

- Hash join partitioning
- External merge sort
- Vectorized batching

Figure 6: Memory Management Architecture

5.3 Vectorized Execution

Modern query engines use vectorized execution for improved performance:

Listing 4: Vectorized Batch Processing

```
struct TupleBatch {
    std::vector<Tuple> tuples;
    std::vector<std::string> column_names;
    size_t batch_size = 1000;  // Configurable batch size

    void add_tuple(const Tuple& tuple) {
        tuples.push_back(tuple);
    }

    bool is_full() const {
        return tuples.size() >= batch_size;
    }
};

class SequentialScanNode : public PhysicalPlanNode {
public:
    TupleBatch get_next_batch() override {
        TupleBatch batch;
        batch.column_names = output_columns;

        // Process tuples in batches for better cache locality
        size_t end_pos = std::min(current_position + batch_size,
                                  mock_data.size());

        for (size_t i = current_position; i < end_pos; ++i) {
            if (passes_filter(mock_data[i])) {
                batch.add_tuple(mock_data[i]);
            }
        }

        current_position = end_pos;
        return batch;
    }
};
```

6 Execution Engine

6.1 Iterator Model

The execution engine implements the iterator model (also known as the Volcano model):

6.2 Parallel Execution

The system supports parallel execution through worker threads:

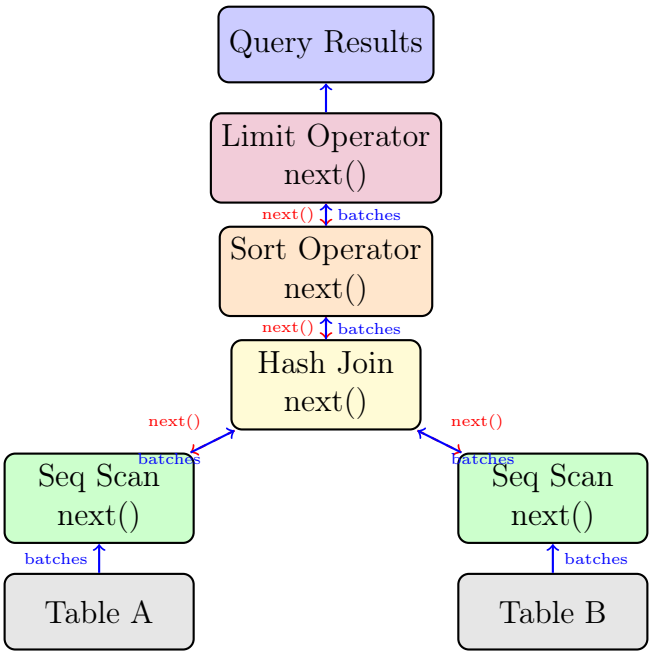


Figure 7: Iterator Model Execution Flow

Algorithm 1 Parallel Sequential Scan Algorithm

Require: Table T, Filter predicate P, Number of workers W
Ensure: Filtered tuples from T

Initialize shared result queue Q
Initialize parallel synchronization context ctx
$rows\_per\_worker \leftarrow \lfloor |T|/W \rfloor$
for $i \leftarrow 0$ **to** $W - 1$ **do**
  $start\_row \leftarrow i \times rows\_per\_worker$
  if $i = W - 1$ **then**
    $end\_row \leftarrow |T|$ {Last worker handles remainder}
  else
    $end\_row \leftarrow (i + 1) \times rows\_per\_worker$
  end if
  LaunchWorkerThread$(i, start_row, end_row, P, Q, ctx)$
end for
while $ctx.active\_workers > 0$ **or** $\neg Q.isEmpty()$ **do**
  if $\neg Q.isEmpty()$ **then**
    $batch \leftarrow Q.dequeue()$
    yield *batch* to parent operator
  end if
end while
JoinAllWorkerThreads()

Algorithm 2 Worker Thread Scan Procedure

Require: Worker ID $worker\_id$, Start row $start$, End row end, Predicate P, Queue Q, Context ctx

$ctx.active\_workers \leftarrow ctx.active\_workers + 1$
Initialize empty batch $batch$
for $row\_idx \leftarrow start$ **to** $end - 1$ **do**
  $tuple \leftarrow T[row\_idx]$
  if EvaluatePredicate$(P, tuple)$ **then**
    $batch.addTuple(tuple)$
    if $batch.isFull()$ **then**
      $Q.enqueue(batch)$
      $batch \leftarrow$ new empty batch
    end if
  end if
end for
if $\neg batch.isEmpty()$ **then**
  $Q.enqueue(batch)$
end if
$ctx.active\_workers \leftarrow ctx.active\_workers - 1$
if $ctx.active\_workers = 0$ **then**
  $ctx.signalCompletion()$
end if

Listing 5: Parallel Execution Implementation

```
class ParallelSequentialScanNode : public PhysicalPlanNode {
    std::shared_ptr<ParallelContext> parallel_ctx;
    std::vector<std::thread> worker_threads;

public:
    void initialize(ExecutionContext* ctx) override {
        parallel_ctx = std::make_shared<ParallelContext>();

        // Start worker threads
        size_t rows_per_worker = mock_data.size() /
           parallel_degree;
        for (size_t i = 0; i < parallel_degree; ++i) {
            size_t start_row = i * rows_per_worker;
            size_t end_row = (i == parallel_degree - 1) ?
                            mock_data.size() : (i + 1) *
                               rows_per_worker;

            worker_threads.emplace_back([this, i, start_row,
               end_row]() {
                worker_scan(i, start_row, end_row);
            });
        }
    }

    TupleBatch get_next_batch() override {
```

```
        return parallel_ctx->get_result_batch();
    }
};
```

7 Current Implementation Status

7.1 Implemented Components

Table 2: Implementation Completeness Matrix

Component	Status	Completeness	Description
SQL Parsing	✓Complete	95%	libpg_query integration
Schema Management	✓Complete	90%	DDL support, validation
Logical Planning	✓Complete	85%	Cost-based optimization
Physical Planning	✓Complete	80%	Operator selection
Basic Execution	✓Complete	75%	Iterator model, batching
Parallel Execution	✓Complete	70%	Worker thread coordination
Mock/Simplified Components			
Data Storage	△ Mock	20%	In-memory mock data
Expression Eval	△ Limited	30%	Basic string matching
Type System	△ Missing	10%	String-based only
Missing Components			
Storage Engine	× Missing	0%	Pages, buffer pool
Transaction Mgmt	× Missing	0%	ACID properties
Index Management	× Missing	0%	B-trees, hash indexes
Concurrency Control	× Missing	0%	Locking, MVCC

7.2 Demonstration Capabilities

The current implementation can successfully execute:

Listing 6: Supported Query Examples

```
// Basic selection and projection
"SELECT * FROM users WHERE id = 123 LIMIT 10"

// Joins with multiple tables
"SELECT u.name, p.name FROM users u JOIN products p ON u.id = p.id
    "
```

```
6
7  // Sorting and limiting
8  "SELECT * FROM users ORDER BY name LIMIT 10"
9
10 // Complex queries with optimization
11 "SELECT u.name, p.name FROM users u JOIN products p ON u.id = p.id
12  WHERE u.name LIKE 'John%' AND p.price > 50"
```

Output includes detailed execution plans and statistics:

Listing 7: Example Execution Plan Output

```
QUERY PLAN
------------------------------------------------------------
Limit (cost=578.68..579.26 rows=10)
  Limit: 10
  Sort (cost=578.68..578.68 rows=10000)
    Sort Key: name NULLS LAST
    Seq Scan on users (cost=0.00..246.48 rows=10000)
------------------------------------------------------------
Execution time: 12.345 ms
Rows processed: 10000
Rows returned: 10
Memory used: 1.2 MB
```

8 Future Implementation Roadmap

This section outlines the systematic approach to extending the current implementation into a production-ready database system.

8.1 Phase 1: Storage Engine Foundation

8.1.1 Buffer Pool Manager

Extension Point 8.1 (Buffer Pool Implementation)**.** *Implement a buffer pool manager to handle page-based storage:*

- ***Page Structure**: Fixed-size pages (typically 8KB)*
- ***Replacement Policy**: LRU or Clock algorithm*
- ***Dirty Page Management**: Write-back caching*
- ***Concurrent Access**: Reader-writer locks*

Listing 8: Buffer Pool Manager Interface

```
1  class BufferPoolManager {
2      struct PageFrame {
3          PageId page_id;
4          char* data;
```

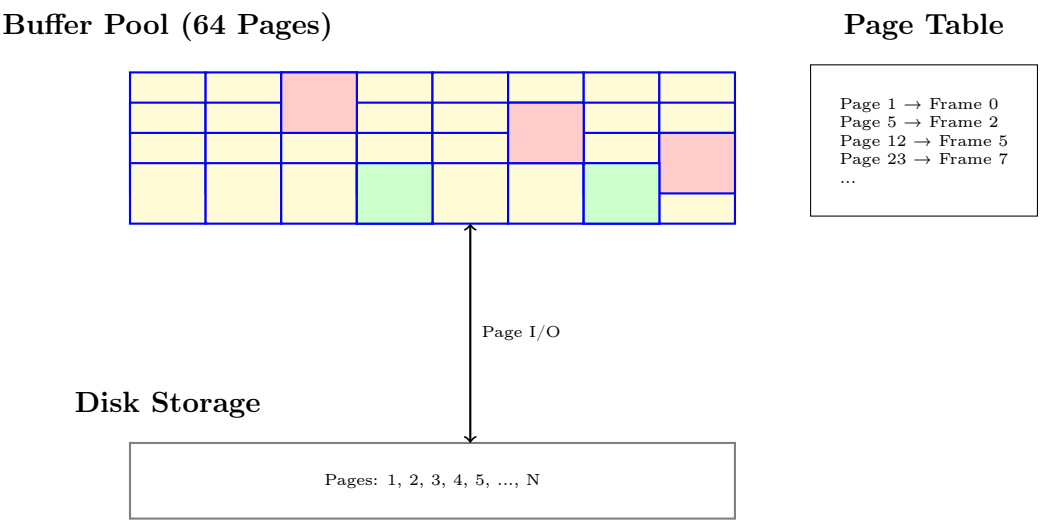


Figure 8: Buffer Pool Manager Architecture

```
        bool is_dirty;
        int pin_count;
        std::chrono::time_point<std::chrono::steady_clock>
           last_access;
    };

    std::vector<PageFrame> frames_;
    std::unordered_map<PageId, FrameId> page_table_;
    std::mutex latch_;

public:
    Page* fetch_page(PageId page_id);
    bool unpin_page(PageId page_id, bool is_dirty);
    Page* new_page(PageId* page_id);
    bool delete_page(PageId page_id);
    void flush_all_pages();

private:
    FrameId find_victim_frame();
    void flush_page(FrameId frame_id);
};
```

8.1.2 Page Management

Extension Point 8.2 (Page Layout Design)**.** *Implement efficient page layouts for different data types:*

- ***Slotted Pages****: Variable-length tuples*
- ***Fixed-Length Records****: High-performance access*
- ***Overflow Pages****: Large attributes (TOAST)*
- ***Free Space Management****: Efficient space utilization*

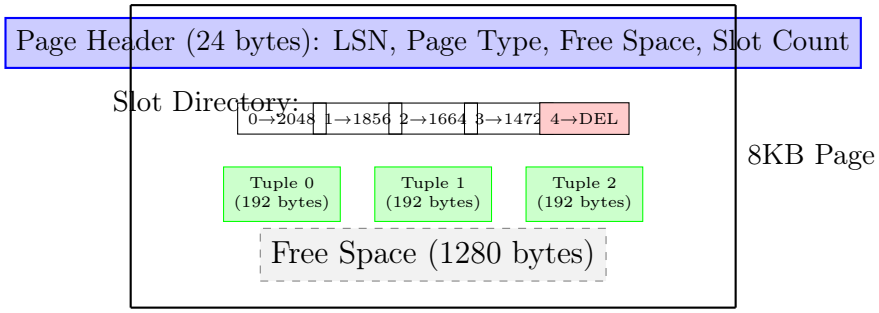


Figure 9: Slotted Page Layout

8.1.3 File Management

Extension Point 8.3 (File System Integration)**.** *Implement file management for persistent storage:*

- ***Heap Files****: Unordered tuple storage*
- ***Directory Pages****: Page allocation tracking*
- ***Extent Management****: Efficient space allocation*
- ***File Growth****: Dynamic file expansion*

8.2 Phase 2: Transaction Management

8.2.1 Transaction Interface

Extension Point 8.4 (Transaction Processing)**.** *Implement ACID transaction support:*

- ***Transaction Context****: State tracking per transaction*
- ***Begin/Commit/Abort****: Transaction lifecycle*
- ***Isolation Levels****: Read uncommitted to serializable*
- ***Deadlock Detection****: Timeout and graph-based*

Listing 9: Transaction Manager Interface

```
class TransactionManager {
    struct Transaction {
        TransactionId txn_id;
        TransactionState state;
        IsolationLevel isolation_level;
        std::chrono::time_point<std::chrono::system_clock>
           start_time;
        std::set<PageId> read_set;
        std::set<PageId> write_set;
    };

    std::unordered_map<TransactionId, Transaction> active_txns_;
    std::atomic<TransactionId> next_txn_id_{1};
```

```
13
14  public:
15      TransactionId begin_transaction(IsolationLevel level =
            IsolationLevel::READ_COMMITTED);
16      void commit_transaction(TransactionId txn_id);
17      void abort_transaction(TransactionId txn_id);
18
19      bool is_transaction_active(TransactionId txn_id);
20      IsolationLevel get_isolation_level(TransactionId txn_id);
21  };
```

8.2.2 Write-Ahead Logging (WAL)

Extension Point 8.5 (Logging System). *Implement write-ahead logging for durability and recovery:*

- ***Log Records**: Before/after images*
- ***Log Sequence Numbers (LSNs)**: Ordering and recovery*
- ***Checkpointing**: Periodic consistency points*
- ***Recovery**: REDO/UNDO processing*

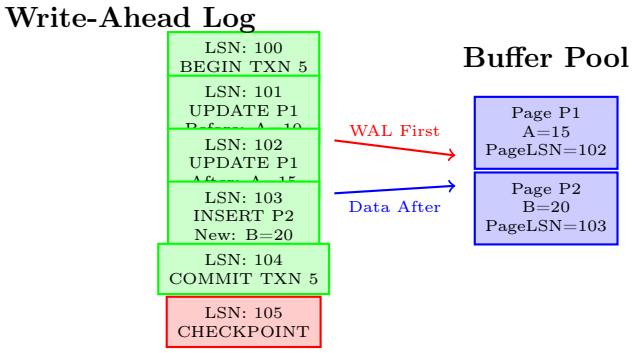


Figure 10: Write-Ahead Logging Protocol

8.3 Phase 3: Concurrency Control

8.3.1 Locking Manager

Extension Point 8.6 (Lock Management). *Implement hierarchical locking for concurrency control:*

- ***Lock Modes**: Shared, Exclusive, Intention locks*
- ***Lock Granularity**: Table, page, tuple-level*
- ***Deadlock Prevention**: Ordering protocols*
- ***Lock Escalation**: Fine to coarse-grained locks*

Table 3: Lock Compatibility Matrix

	IS	IX	S	X	SIX
IS	✓	✓	✓	×	✓
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
X	×	×	×	×	×
SIX	✓	×	×	×	×

8.3.2 Multi-Version Concurrency Control (MVCC)

Extension Point 8.7 (MVCC Implementation)**.** *Implement MVCC for improved concurrency:*

- ***Tuple Versioning****: Multiple tuple versions*
- ***Visibility Rules****: Transaction snapshot isolation*
- ***Garbage Collection****: Old version cleanup*
- ***Version Chains****: Linked list of versions*

8.4 Phase 4: Index Management

8.4.1 B+ Tree Implementation

Extension Point 8.8 (B+ Tree Index)**.** *Implement B+ tree indexes for efficient data access:*

- ***Node Structure****: Internal and leaf nodes*
- ***Insertion/Deletion****: Tree balancing algorithms*
- ***Range Queries****: Efficient scan operations*
- ***Concurrent Access****: Latch coupling protocol*

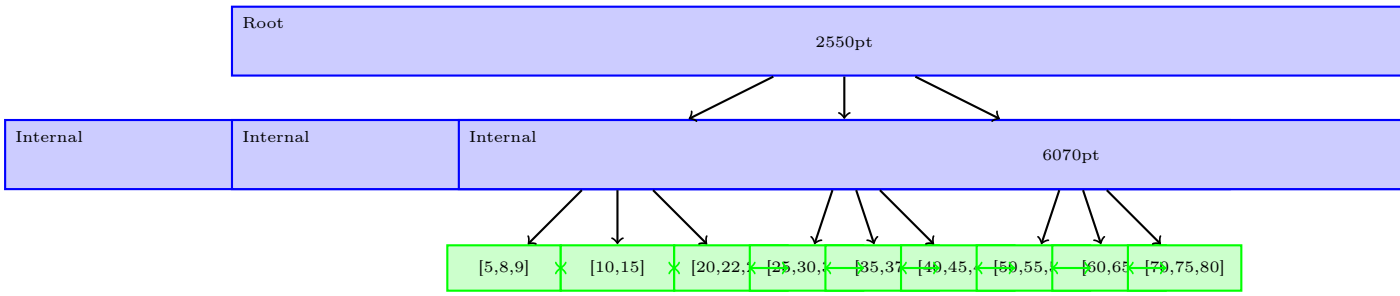


Figure 11: B+ Tree Index Structure

8.4.2 Hash Indexes

Extension Point 8.9 (Hash Index Implementation)**.** *Implement hash indexes for equality queries:*

- ***Extendible Hashing****: Dynamic bucket splitting*
- ***Collision Handling****: Chaining or open addressing*
- ***Hash Functions****: Uniform distribution*
- ***Rehashing****: Load factor management*

8.5 Phase 5: Advanced Query Processing

8.5.1 Complex Expression Evaluation

Extension Point 8.10 (Expression Engine)**.** *Implement a comprehensive expression evaluation system:*

- ***Type System****: Strong typing with conversions*
- ***Function Library****: Built-in SQL functions*
- ***Operator Precedence****: Correct expression parsing*
- ***Null Handling****: Three-valued logic*

Listing 10: Expression Evaluation Framework

```
class Expression {
public:
    virtual ~Expression() = default;
    virtual Value evaluate(const Tuple& tuple, ExecutionContext&
       ctx) = 0;
    virtual DataType get_return_type() const = 0;
    virtual std::unique_ptr<Expression> clone() const = 0;
};

class BinaryOpExpression : public Expression {
    std::unique_ptr<Expression> left_;
    std::unique_ptr<Expression> right_;
    BinaryOpType op_type_;

public:
    Value evaluate(const Tuple& tuple, ExecutionContext& ctx)
       override {
        Value left_val = left_->evaluate(tuple, ctx);
        Value right_val = right_->evaluate(tuple, ctx);

        return apply_binary_op(left_val, right_val, op_type_);
    }
};
```

```
22
23 class FunctionExpression : public Expression {
24     std::string function_name_;
25     std::vector<std::unique_ptr<Expression>> arguments_;
26
27 public:
28     Value evaluate(const Tuple& tuple, ExecutionContext& ctx)
       override {
29         std::vector<Value> arg_values;
30         for (auto& arg : arguments_) {
31             arg_values.push_back(arg->evaluate(tuple, ctx));
32         }
33
34         return ctx.function_registry.call(function_name_,
           arg_values);
35     }
36 };
```

8.5.2 Advanced Join Algorithms

Extension Point 8.11 (Join Algorithm Enhancement)**.** *Implement additional join algorithms:*

- ***Sort-Merge Join****: For pre-sorted inputs*
- ***Hybrid Hash Join****: Memory-adaptive partitioning*
- ***Index Nested Loop****: Using index lookups*
- ***Multi-Way Joins****: Star and snowflake queries*

8.5.3 Advanced Aggregation

Extension Point 8.12 (Aggregation Enhancement)**.** *Implement sophisticated aggregation operators:*

- ***Window Functions****: OVER clauses*
- ***CUBE and ROLLUP****: Multi-dimensional aggregation*
- ***Streaming Aggregation****: Large dataset processing*
- ***Approximate Aggregation****: HyperLogLog, sketches*

8.6 Phase 6: Advanced Optimization

8.6.1 Statistics and Cardinality Estimation

Extension Point 8.13 (Statistics System)**.** *Implement comprehensive statistics for query optimization:*

- ***Histograms****: Value distribution tracking*

Algorithm 3 Hash Join Algorithm

Require: Left relation R, Right relation S, Join predicate θ
Ensure: Joined tuples satisfying θ

Phase 1: Build Phase
Initialize hash table H
for each tuple $r \in R$ **do**
  $key \leftarrow \text{ExtractJoinKey}(r, \theta)$
  $H[key].append(r)$ {Add to hash bucket}
end for
Phase 2: Probe Phase
for each tuple $s \in S$ **do**
  $key \leftarrow \text{ExtractJoinKey}(s, \theta)$
  if $key \in H$ **then**
    for each tuple $r \in H[key]$ **do**
      if $\text{EvaluateJoinCondition}(r, s, \theta)$ **then**
        yield $\text{MergeTuples}(r, s)$
      end if
    end for
  end if
end for

Algorithm 4 External Sort Algorithm

Require: Input relation R, Available memory M, Sort keys K
Ensure: Sorted relation R'

Phase 1: Generate Sorted Runs
$run\_count \leftarrow 0$
$buffer \leftarrow$ empty list
while R has more tuples **do**
  Fill $buffer$ with up to M tuples from R
  $\text{InMemorySort}(buffer, K)$
  Write $buffer$ to temporary file $temp\_run_{run\_count}$
  $run\_count \leftarrow run\_count + 1$
  Clear $buffer$
end while
Phase 2: Merge Sorted Runs
Initialize priority queue PQ with first tuple from each run
Open output file R'
while PQ is not empty **do**
  $(tuple, run\_id) \leftarrow PQ.extractMin()$
  Write $tuple$ to R'
  if $temp\_run_{run\_id}$ has more tuples **then**
    $next\_tuple \leftarrow$ read next tuple from $temp\_run_{run\_id}$
    $PQ.insert((next\_tuple, run\_id))$
  end if
end while
Delete all temporary run files
return R'

- ***Most Common Values (MCVs)****: Skew handling*
- ***Correlation Statistics****: Multi-column dependencies*
- ***Adaptive Statistics****: Query feedback integration*

8.6.2 Advanced Cost Models

Extension Point 8.14 (Cost Model Enhancement)**.** *Develop sophisticated cost estimation:*

- ***Machine Learning****: Learned cost models*
- ***Runtime Feedback****: Actual vs estimated costs*
- ***Hardware-Aware Costs****: CPU, memory, I/O modeling*
- ***Parallel Cost Models****: Multi-threading overhead*

9 Teaching Methodology

9.1 Progressive Implementation Approach

This implementation framework supports a structured learning approach:

1. **Phase 1 - Foundations**: Students begin with the current working system
2. **Phase 2 - Storage**: Implement file and page management
3. **Phase 3 - Transactions**: Add ACID properties
4. **Phase 4 - Concurrency**: Implement locking and MVCC
5. **Phase 5 - Indexing**: Add B+ trees and hash indexes
6. **Phase 6 - Advanced**: Optimize and extend functionality

9.2 Learning Objectives by Phase

9.3 Assessment Strategies

- **Incremental Development**: Each phase builds on previous work
- **Performance Benchmarking**: Measure improvements at each stage
- **Design Documentation**: Require architectural documentation
- **Testing Framework**: Comprehensive test suite development
- **Research Extensions**: Open-ended optimization projects

Table 4: Learning Objectives by Implementation Phase

Phase	Technical Skills	Conceptual Understanding
Current	C++17, SQL parsing, query planning	Query optimization theory
Storage	File I/O, memory management, caching	Storage hierarchy, buffer management
Transactions	Logging, recovery, state management	ACID properties, consistency
Concurrency	Threading, synchronization, deadlocks	Isolation levels, conflict serializability
Indexing	Tree algorithms, hashing, B+ trees	Access methods, query performance
Advanced	Performance tuning, statistics	Research-level optimization

10 Integration with Current Implementation

10.1 Extension Points

The current architecture provides clear extension points:

Listing 11: Storage Interface Extension Point

```
// Current mock implementation
class SequentialScanNode : public PhysicalPlanNode {
    std::vector<Tuple> mock_data;  // Replace with storage
        interface

public:
    TupleBatch get_next_batch() override {
        // Current: iterate over mock_data
        // Future: integrate with buffer pool manager

        TupleBatch batch;
        // TODO: Replace with real storage access
        for (size_t i = current_position; i < end_pos; ++i) {
            if (passes_filter(mock_data[i])) {
                batch.add_tuple(mock_data[i]);
            }
        }
        return batch;
    }
};

// Future storage-integrated implementation
class StorageSequentialScanNode : public PhysicalPlanNode {
    TableOid table_oid;
    std::shared_ptr<BufferPoolManager> buffer_pool_;
```

```
    std::shared_ptr<CatalogManager> catalog_;

public:
    TupleBatch get_next_batch() override {
        TupleBatch batch;

        // Get table metadata from catalog
        auto table_info = catalog_->get_table_info(table_oid);

        // Iterate through pages using buffer pool
        while (current_page_id <= table_info->last_page_id) {
            Page* page = buffer_pool_->fetch_page(current_page_id);

            // Extract tuples from page
            auto page_tuples = extract_tuples_from_page(page,
                table_info->schema);

            for (const auto& tuple : page_tuples) {
                if (passes_filter(tuple)) {
                    batch.add_tuple(tuple);
                    if (batch.is_full()) break;
                }
            }

            buffer_pool_->unpin_page(current_page_id, false);
            if (batch.is_full()) break;

            current_page_id++;
        }

        return batch;
    }
};
```

10.2 Backward Compatibility

Extensions maintain compatibility with existing interfaces:

Listing 12: Interface Compatibility Design

```
// Abstract base maintains current interface
class PhysicalPlanNode {
public:
    virtual TupleBatch get_next_batch() = 0;
    virtual void reset() = 0;
    // Interface remains stable
};

// Extensions add new capabilities
class StorageAwarePhysicalPlanNode : public PhysicalPlanNode {
protected:
```

```
    std::shared_ptr<StorageManager> storage_manager_;
    std::shared_ptr<TransactionManager> txn_manager_;

public:
    // Existing interface
    TupleBatch get_next_batch() override = 0;
    void reset() override = 0;

    // New storage-aware methods
    virtual void set_storage_manager(std::shared_ptr<
       StorageManager> sm) {
        storage_manager_ = sm;
    }

    virtual void set_transaction_context(TransactionId txn_id) {
        current_txn_id_ = txn_id;
    }
};
```

11 Performance Analysis and Benchmarking

11.1 Current Performance Characteristics

The existing implementation provides a baseline for performance comparison:

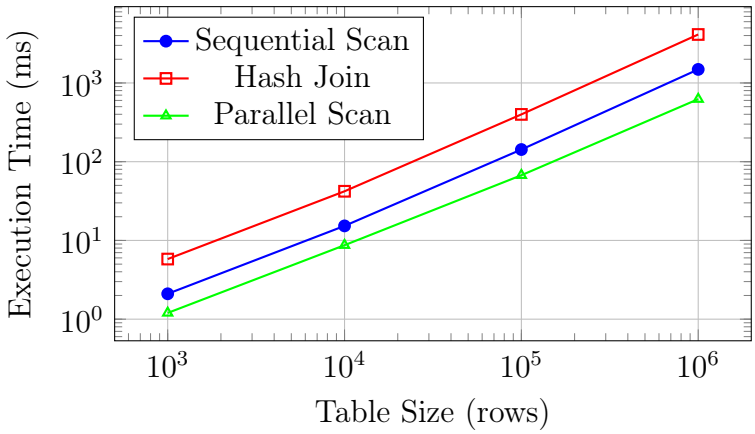


Figure 12: Performance Scaling with Mock Data

11.2 Benchmarking Framework

Extension Point 11.1 (Performance Testing)**.** *Implement comprehensive benchmarking:*

- ***TPC Benchmarks****: TPC-H for analytical queries*
- ***Microbenchmarks****: Individual operator performance*
- ***Scalability Tests****: Multi-threaded performance*
- ***Memory Usage Analysis****: Resource consumption tracking*

11.3 Expected Performance Improvements

Performance improvements anticipated from each phase:

Table 5: Expected Performance Gains by Implementation Phase

Phase	Improvement	Mechanism	Workload Impact
Storage Engine	10-100x	Real I/O optimization, caching	Large dataset queries
Indexing	100-1000x	B+ tree access	Selective queries
Transactions	Varies	Reduced locking overhead	Concurrent workloads
MVCC	2-10x	Reduced blocking	Read-heavy workloads
Vectorization	2-5x	SIMD, cache optimization	CPU-intensive queries
Parallelization	2-8x	Multi-core utilization	Large scan operations

12 Research Extensions and Future Work

12.1 Machine Learning Integration

Extension Point 12.1 (ML-Enhanced Query Processing)**.** *Integrate machine learning for intelligent query processing:*

- ***Learned Indexes****: Replace B+ trees with learned models*
- ***Cardinality Estimation****: Neural network-based estimates*
- ***Join Order Optimization****: Reinforcement learning*
- ***Adaptive Query Processing****: Runtime plan adjustments*

12.2 Modern Hardware Utilization

Extension Point 12.2 (Hardware-Aware Processing)**.** *Optimize for modern hardware architectures:*

- ***SIMD Vectorization****: AVX-512 instruction utilization*
- ***GPU Acceleration****: CUDA-based query processing*
- ***Non-Volatile Memory****: Persistent memory integration*
- ***RDMA Networks****: High-speed interconnects*

12.3 Distributed Query Processing

Extension Point 12.3 (Distributed Systems)**.** *Extend to distributed query processing:*

- ***Data Partitioning****: Horizontal and vertical partitioning*
- ***Distributed Joins****: Cross-node join processing*
- ***Consensus Protocols****: Distributed transaction coordination*
- ***Fault Tolerance****: Node failure recovery*

13 Conclusion

This comprehensive implementation framework provides a solid foundation for understanding and extending modern query processing systems. The current implementation demonstrates core concepts while providing clear pathways for production-ready enhancements.

13.1 Key Contributions

1. **Complete Pipeline**: End-to-end query processing implementation
2. **Educational Framework**: Structured learning progression
3. **Extension Architecture**: Clear interfaces for enhancements
4. **Modern Techniques**: Vectorization and parallel processing
5. **Research Integration**: Academic and industrial practices

13.2 Learning Outcomes

Students working with this system will gain:

- Deep understanding of query processing internals
- Practical experience with database system implementation
- Exposure to modern optimization techniques
- Foundation for database research and development
- Skills applicable to distributed and cloud systems

The systematic approach outlined in this paper enables both academic instruction and research advancement, providing a bridge between theoretical database concepts and practical system implementation.

14 Acknowledgments

This implementation builds upon decades of database systems research and the open-source PostgreSQL project. Special recognition goes to the libpg_query project for providing accessible PostgreSQL parsing capabilities.

References

[1] Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database Systems: The Complete Book*. Pearson Prentice Hall.

[2] Ramakrishnan, R., & Gehrke, J. (2003). *Database Management Systems*. McGraw-Hill.

[3] Hellerstein, J. M., Stonebraker, M., & Hamilton, J. (2007). Architecture of a database system. *Foundations and Trends in Databases*, 1(2), 141-259.

[4] Graefe, G. (1993). Volcano—an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1), 120-135.

[5] Neumann, T. (2011). Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9), 539-550.

[6] PostgreSQL Global Development Group. (2023). PostgreSQL Documentation. `https://www.postgresql.org/docs/`

[7] libpg_query. (2023). Standalone PostgreSQL query parser for C/C++. `https://github.com/pganalyze/libpg_query`