

DB25: Modern Query Processing Engine Architecture

A Comprehensive Implementation and Extension Framework

Chiradip Mandal
chiradip@chiradip.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	4
1.1	System Overview	4
1.2	Educational Objectives	4
1.3	Algorithmic Notation	5
2	System Architecture	5
2.1	Component Hierarchy	5
2.2	Core Classes and Interfaces	7
3	Query Parsing and Validation	7
3.1	Integration with libpg_query	7
3.2	AST Processing	7

4	Logical Query Planning	9
4.1	Plan Node Types	9
4.2	Cost Model	9
4.3	Query Optimization Rules	10
5	Physical Query Planning	10
5.1	Operator Selection	10
5.2	Memory Management	11
5.3	Vectorized Execution	11
6	Execution Engine	12
6.1	Iterator Model	12
6.2	Parallel Execution	13
7	Current Implementation Status	15
7.1	Implemented Components	15
7.2	Demonstration Capabilities	15
8	Future Implementation Roadmap	16
8.1	Phase 1: Storage Engine Foundation	16
8.1.1	Buffer Pool Manager	16
8.1.2	Page Management	17
8.1.3	File Management	18
8.2	Phase 2: Transaction Management	18
8.2.1	Transaction Interface	18
8.2.2	Write-Ahead Logging (WAL)	19
8.3	Phase 3: Concurrency Control	19
8.3.1	Locking Manager	19
8.3.2	Multi-Version Concurrency Control (MVCC)	20
8.4	Phase 4: Index Management	20
8.4.1	B+ Tree Implementation	20
8.4.2	Hash Indexes	21
8.5	Phase 5: Advanced Query Processing	21
8.5.1	Complex Expression Evaluation	21
8.5.2	Advanced Join Algorithms	22
8.5.3	Advanced Aggregation	22
8.6	Phase 6: Advanced Optimization	24
8.6.1	Statistics and Cardinality Estimation	24
8.6.2	Advanced Cost Models	24
9	Teaching Methodology	24
9.1	Progressive Implementation Approach	24
9.2	Learning Objectives by Phase	25
9.3	Assessment Strategies	25
10	Integration with Current Implementation	25
10.1	Extension Points	25
10.2	Backward Compatibility	27

11 Performance Analysis and Benchmarking	27
11.1 Current Performance Characteristics	27
11.2 Benchmarking Framework	27
11.3 Expected Performance Improvements	28
12 Research Extensions and Future Work	28
12.1 Machine Learning Integration	28
12.2 Modern Hardware Utilization	29
12.3 Distributed Query Processing	29
13 Conclusion	29
13.1 Key Contributions	29
13.2 Learning Outcomes	29
14 Acknowledgments	30

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

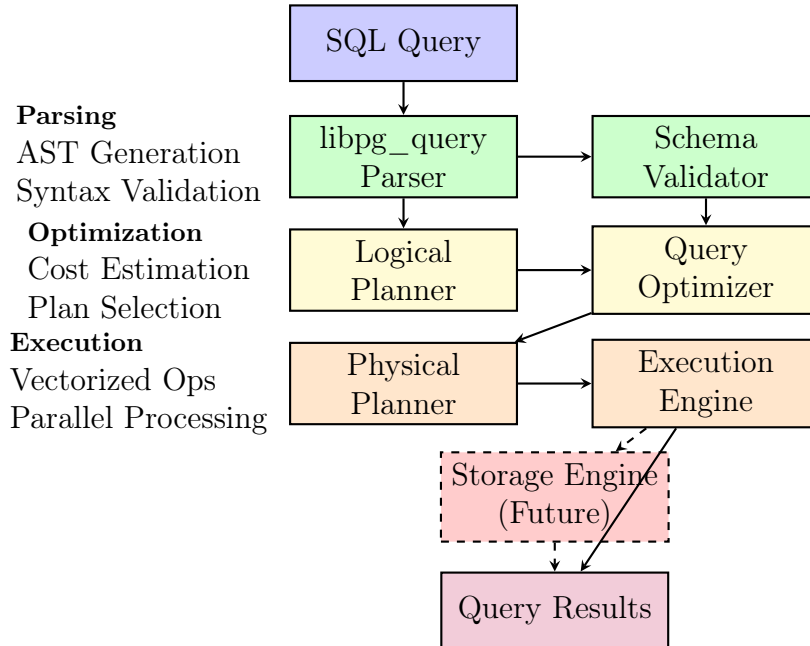


Figure 1: Query Processing Architecture Overview

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:

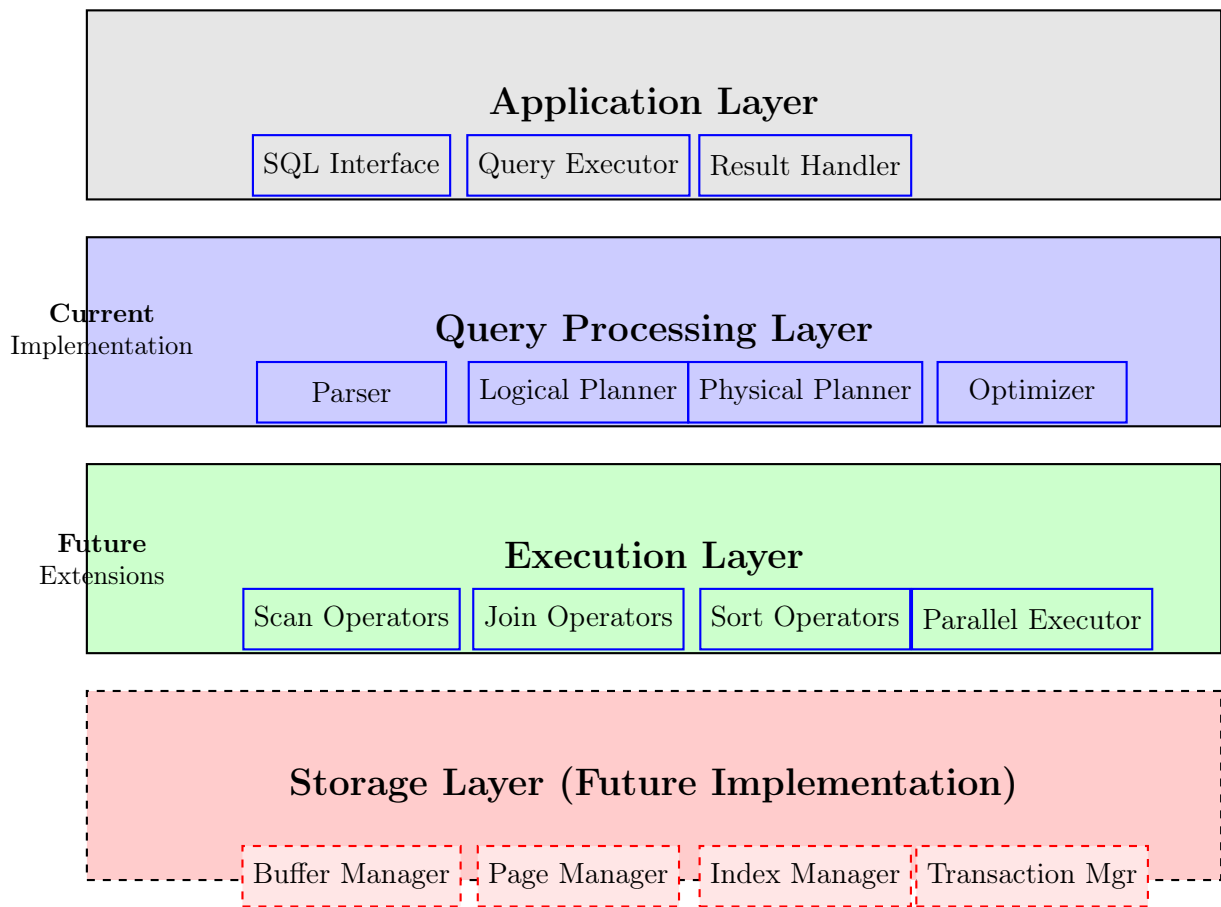


Figure 2: System Component Hierarchy

2.2 Core Classes and Interfaces

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

Listing 1: Core Interface Definitions

```
1 namespace pg {
2     // Base query plan node
3     struct LogicalPlanNode {
4         PlanNodeType type;
5         PlanCost cost;
6         std::vector<LogicalPlanNodePtr> children;
7         std::vector<std::string> output_columns;
8
9         virtual std::string to_string(int indent = 0) const = 0;
10        virtual LogicalPlanNodePtr copy() const = 0;
11    };
12
13    // Physical execution interface
14    struct PhysicalPlanNode {
15        PhysicalOperatorType type;
16        ExecutionStats actual_stats;
17
18        virtual TupleBatch get_next_batch() = 0;
19        virtual void reset() = 0;
20        virtual void initialize(ExecutionContext* ctx) = 0;
21    };
22
23    // Main planner interface
24    class QueryPlanner {
25        std::shared_ptr<DatabaseSchema> schema_;
26        CostModel cost_model_;
27
28    public:
29        LogicalPlan create_plan(const std::string& query);
30        std::vector<LogicalPlan> generate_alternatives(const std::string& query);
31        void optimize_plan(LogicalPlan& plan);
32    };
33 }
```

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:

3.2 AST Processing

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

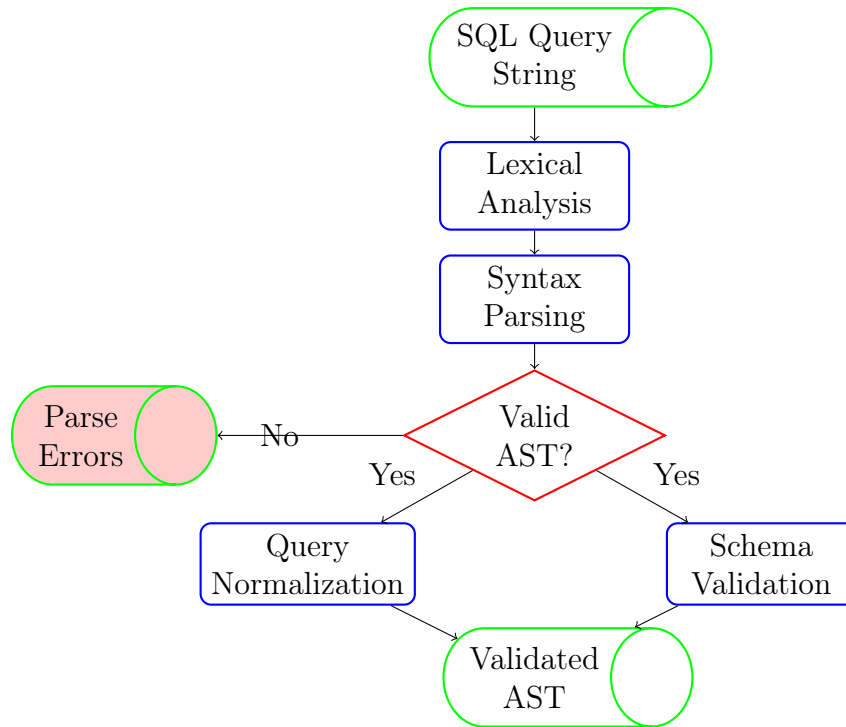


Figure 3: SQL Parsing Pipeline

Listing 2: AST Processing Example

```

1  class PgQueryWrapper {
2      ParseResult parse(const std::string& query) {
3          ParseResult result;
4
5          // Use libpg_query for parsing
6          auto pg_result = pg_query_parse(query.c_str());
7
8          if (pg_result.error) {
9              result.is_valid = false;
10             result.errors.push_back(pg_result.error->message);
11         } else {
12             result.is_valid = true;
13             result.parse_tree = pg_result.parse_tree;
14
15             // Extract query components
16             extract_table_references(pg_result.parse_tree, result)
17             ;
18             extract_column_references(pg_result.parse_tree, result
19             );
20         }
21
22         pg_query_free_parse_result(pg_result);
23         return result;
24     }
25 };

```


4 Logical Query Planning

4.1 Plan Node Types

The logical planning phase creates a tree of logical operators:

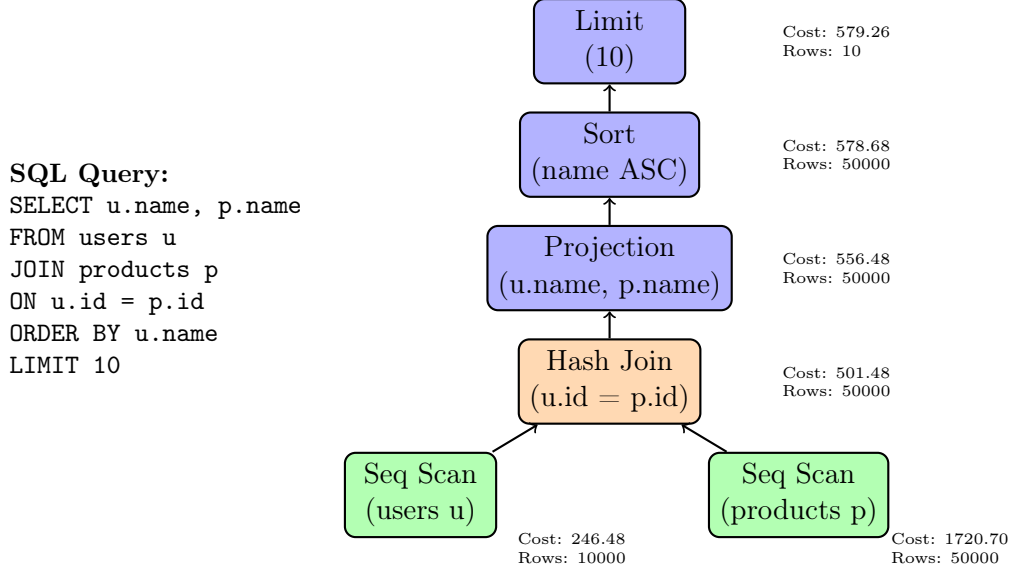


Figure 4: Logical Query Plan Tree Structure

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:*

$$C_{total}(N) = C_{startup}(N) + C_{run}(N) \quad (1)$$

$$C_{run}(N) = C_{cpu}(N) + C_{io}(N) \quad (2)$$

$$C_{cpu}(N) = rows(N) \times c_{cpu} \quad (3)$$

$$C_{io}(N) = pages(N) \times c_{io} \quad (4)$$

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

Listing 3: Cost Calculation Implementation

```

1 struct PlanCost {
2     double startup_cost = 0.0;
3     double total_cost = 0.0;
4     size_t estimated_rows = 0;
5     double estimated_width = 0.0;
6
7     // Cost calculation for sequential scan
8     static PlanCost calculate_seq_scan_cost(const TableStats&
9         stats) {
10         PlanCost cost;

```

```

10     cost.startup_cost = 0.0;
11
12     // IO cost: pages * seq_page_cost
13     double io_cost = stats.pages * SEQ_PAGE_COST;
14
15     // CPU cost: tuples * cpu_tuple_cost
16     double cpu_cost = stats.row_count * CPU_TUPLE_COST;
17
18     cost.total_cost = cost.startup_cost + io_cost + cpu_cost;
19     cost.estimated_rows = stats.row_count;
20     cost.estimated_width = stats.avg_row_size;
21
22     return cost;
23 }
24 };

```

4.3 Query Optimization Rules

The optimizer applies transformation rules to improve query plans:

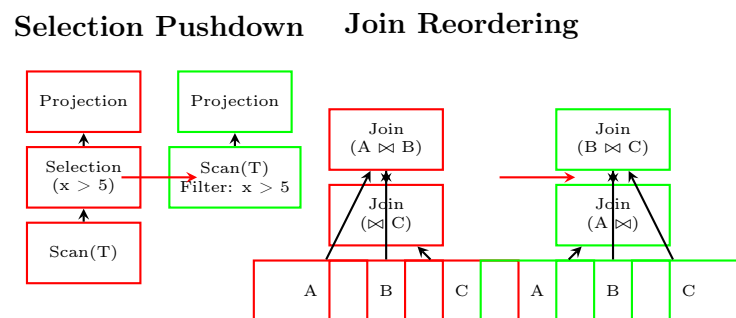


Figure 5: Query Optimization Transformations

5 Physical Query Planning

5.1 Operator Selection

Physical planning converts logical operators into executable physical operators:

Table 1: Logical to Physical Operator Mapping

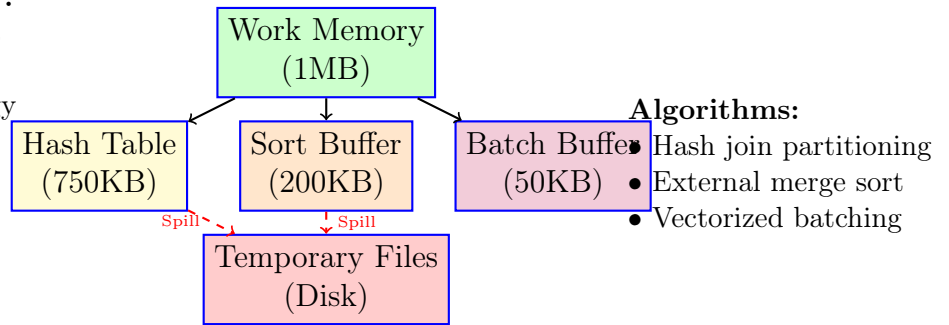
Logical Operator	Physical Options	Selection Criteria
Table Scan	Sequential Scan	Default choice
	Index Scan	Selective predicates
	Parallel Seq Scan	Large tables
Join	Nested Loop Join	Small tables
	Hash Join	One small, one large table
	Sort-Merge Join	Both inputs sorted
Aggregation	Hash Aggregate	GROUP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

5.2 Memory Management

The execution engine implements sophisticated memory management:

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

```

1 struct TupleBatch {
2     std::vector<Tuple> tuples;
3     std::vector<std::string> column_names;
4     size_t batch_size = 1000;    // Configurable batch size
5
6     void add_tuple(const Tuple& tuple) {
7         tuples.push_back(tuple);
8     }
9
10    bool is_full() const {
11        return tuples.size() >= batch_size;
12    }
13 };
14
15 class SequentialScanNode : public PhysicalPlanNode {
16 public:
17     TupleBatch get_next_batch() override {
18         TupleBatch batch;
19         batch.column_names = output_columns;
20
21         // Process tuples in batches for better cache locality
22         size_t end_pos = std::min(current_position + batch_size
23                                     ,
24                                     mock_data.size());
25
26         for (size_t i = current_position; i < end_pos; ++i) {
27             if (passes_filter(mock_data[i])) {
28                 batch.add_tuple(mock_data[i]);
29             }
30         }
31
32         current_position = end_pos;
33         return batch;
34     };

```

6 Execution Engine

6.1 Iterator Model

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

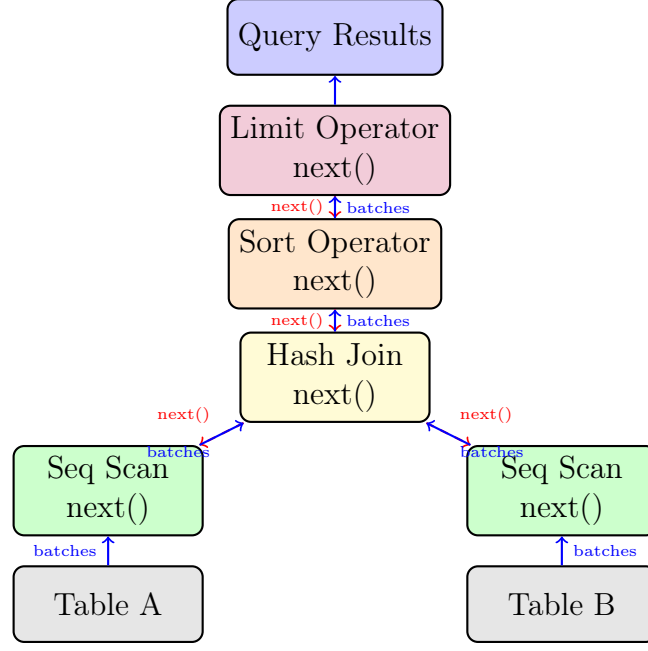


Figure 7: Iterator Model Execution Flow

6.2 Parallel Execution

The system supports parallel execution through worker threads:

Algorithm 1 Parallel Sequential Scan Algorithm

Require: Table T , Filter predicate P , Number of workers W

Ensure: Filtered tuples from T

```

1: Initialize shared result queue  $Q$ 
2: Initialize parallel synchronization context  $ctx$ 
3:  $rows\_per\_worker \leftarrow \lfloor |T|/W \rfloor$ 
4: for  $i \leftarrow 0$  to  $W - 1$  do
5:    $start\_row \leftarrow i \times rows\_per\_worker$ 
6:   if  $i = W - 1$  then
7:      $end\_row \leftarrow |T|$  {Last worker handles remainder}
8:   else
9:      $end\_row \leftarrow (i + 1) \times rows\_per\_worker$ 
10:  end if
11:  LAUNCHWORKERTHREAD( $i, start\_row, end\_row, P, Q, ctx$ )
12: end for
13: while  $ctx.active\_workers > 0$  or  $\neg Q.isEmpty()$  do
14:   if  $\neg Q.isEmpty()$  then
15:      $batch \leftarrow Q.dequeue()$ 
16:     yield  $batch$  to parent operator
17:   end if
18: end while
19: JOINALLWORKERTHREADS()

```

Algorithm 2 Worker Thread Scan Procedure

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

```
1:  $ctx.active\_workers \leftarrow ctx.active\_workers + 1$ 
2: Initialize empty batch  $batch$ 
3: for  $row\_idx \leftarrow start$  to  $end - 1$  do
4:    $tuple \leftarrow T[row\_idx]$ 
5:   if EVALUATEPREDICATE( $P, tuple$ ) then
6:      $batch.addTuple(tuple)$ 
7:     if  $batch.isFull()$  then
8:        $Q.enqueue(batch)$ 
9:        $batch \leftarrow$  new empty batch
10:    end if
11:  end if
12: end for
13: if  $\neg batch.isEmpty()$  then
14:    $Q.enqueue(batch)$ 
15: end if
16:  $ctx.active\_workers \leftarrow ctx.active\_workers - 1$ 
17: if  $ctx.active\_workers = 0$  then
18:    $ctx.signalCompletion()$ 
19: end if
```

Listing 5: Parallel Execution Implementation

```
1  class ParallelSequentialScanNode : public PhysicalPlanNode {
2      std::shared_ptr<ParallelContext> parallel_ctx;
3      std::vector<std::thread> worker_threads;
4
5  public:
6      void initialize(ExecutionContext* ctx) override {
7          parallel_ctx = std::make_shared<ParallelContext>();
8
9          // Start worker threads
10         size_t rows_per_worker = mock_data.size() /
11             parallel_degree;
12         for (size_t i = 0; i < parallel_degree; ++i) {
13             size_t start_row = i * rows_per_worker;
14             size_t end_row = (i == parallel_degree - 1) ?
15                 mock_data.size() : (i + 1) *
16                 rows_per_worker;
17
18             worker_threads.emplace_back([this, i, start_row,
19                 end_row]() {
20                 worker_scan(i, start_row, end_row);
21             });
22         }
23
24         TupleBatch get_next_batch() override {
25             return parallel_ctx->get_result_batch();
26         }
27     };
28 }
```

14

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
<i>Mock/Simplified Components</i>			
Data Storage	△ Mock	20%	In-memory mock data
Expression Eval	△ Limited	30%	Basic string matching
Type System	△ Missing	10%	String-based only
<i>Missing Components</i>			
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

```

1 // Basic selection and projection
2 "SELECT * FROM users WHERE id = 123 LIMIT 10"
3
4 // Joins with multiple tables
5 "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*

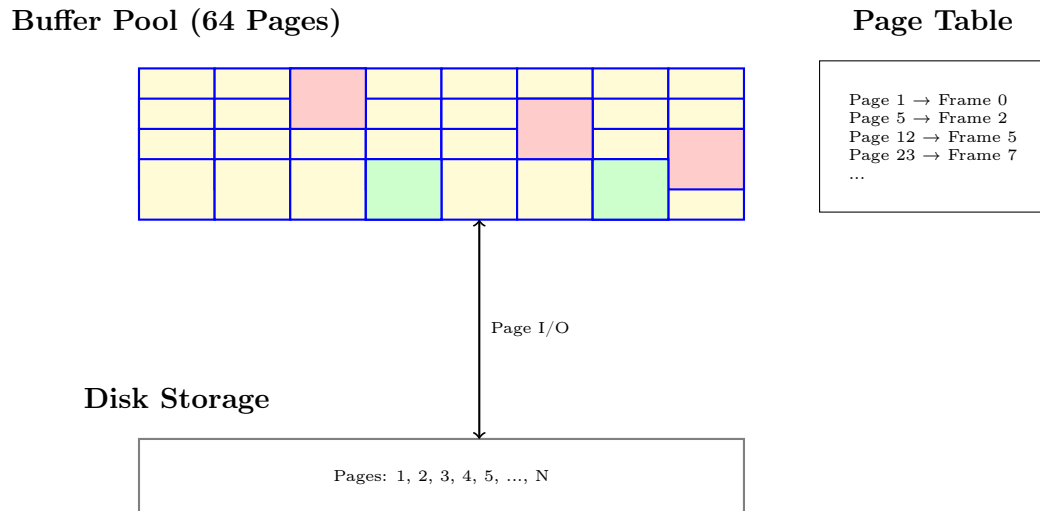


Figure 8: Buffer Pool Manager Architecture

Listing 8: Buffer Pool Manager Interface

```

1 class BufferPoolManager {
2     struct PageFrame {
3         PageId page_id;
4         char* data;
5         bool is_dirty;
6         int pin_count;
7         std::chrono::time_point<std::chrono::steady_clock>
            last_access;
8     };
9
10    std::vector<PageFrame> frames_;
11    std::unordered_map<PageId, FrameId> page_table_;
12    std::mutex latch_;
13
14 public:
15     Page* fetch_page(PageId page_id);
16     bool unpin_page(PageId page_id, bool is_dirty);
17     Page* new_page(PageId* page_id);
18     bool delete_page(PageId page_id);
19     void flush_all_pages();
20
21 private:
22     FrameId find_victim_frame();
23     void flush_page(FrameId frame_id);
24 };

```

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*

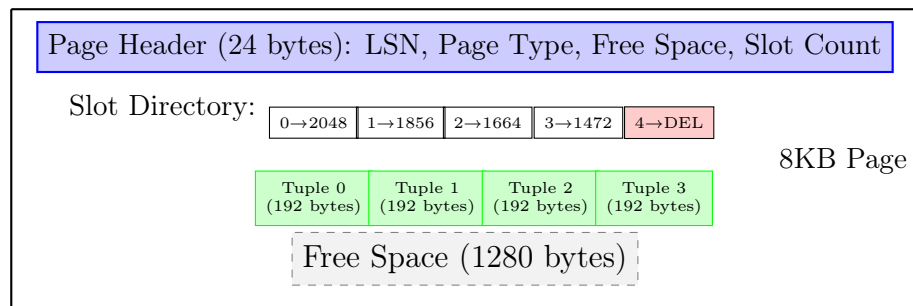


Figure 9: Slotted Page Layout

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

```

1 class TransactionManager {
2     struct Transaction {
3         TransactionId txn_id;
4         TransactionState state;
5         IsolationLevel isolation_level;

```

```

6      std::chrono::time_point<std::chrono::system_clock>
          start_time;
7      std::set<PageId> read_set;
8      std::set<PageId> write_set;
9  };
10
11      std::unordered_map<TransactionId, Transaction> active_txns_;
12      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

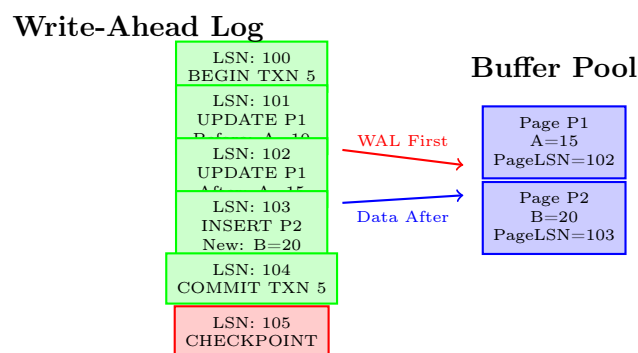


Figure 10: Write-Ahead Logging Protocol

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*

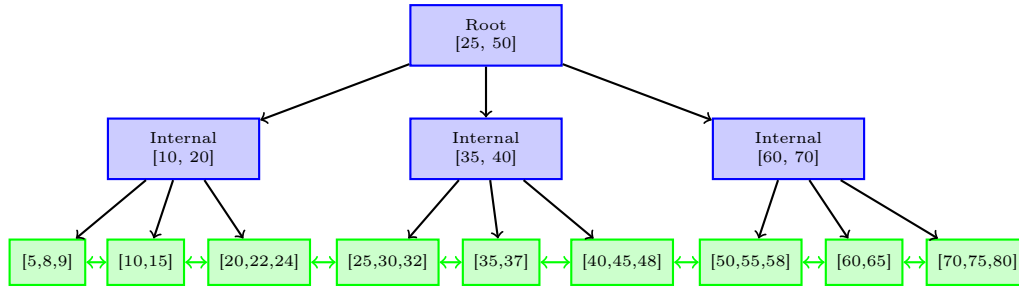


Figure 11: B+ Tree Index Structure

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

```

1 class Expression {
2 public:
3     virtual ~Expression() = default;
4     virtual Value evaluate(const Tuple& tuple, ExecutionContext&
5         ctx) = 0;
6     virtual DataType get_return_type() const = 0;
7     virtual std::unique_ptr<Expression> clone() const = 0;
8 };
9
10 class BinaryOpExpression : public Expression {
11     std::unique_ptr<Expression> left_;
12     std::unique_ptr<Expression> right_;

```

```

12     BinaryOpType op_type_;
13
14 public:
15     Value evaluate(const Tuple& tuple, ExecutionContext& ctx)
16         override {
17         Value left_val = left_->evaluate(tuple, ctx);
18         Value right_val = right_->evaluate(tuple, ctx);
19
20         return apply_binary_op(left_val, right_val, op_type_);
21     };
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)
29         override {
30         std::vector<Value> arg_values;
31         for (auto& arg : arguments_) {
32             arg_values.push_back(arg->evaluate(tuple, ctx));
33         }
34
35         return ctx.function_registry.call(function_name_,
36             arg_values);
37     };

```

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*

Algorithm 3 Hash Join Algorithm

Require: Left relation R , Right relation S , Join predicate θ

Ensure: Joined tuples satisfying θ

```
1: Phase 1: Build Phase
2: Initialize hash table  $H$ 
3: for each tuple  $r \in R$  do
4:    $key \leftarrow \text{EXTRACTJOINKEY}(r, \theta)$ 
5:    $H[key].append(r)$  {Add to hash bucket}
6: end for
7: Phase 2: Probe Phase
8: for each tuple  $s \in S$  do
9:    $key \leftarrow \text{EXTRACTJOINKEY}(s, \theta)$ 
10:  if  $key \in H$  then
11:    for each tuple  $r \in H[key]$  do
12:      if  $\text{EVALUATEJOINCONDITION}(r, s, \theta)$  then
13:        yield  $\text{MERGETUPLES}(r, s)$ 
14:      end if
15:    end for
16:  end if
17: end for
```

Algorithm 4 External Sort Algorithm

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

Ensure: Sorted relation R'

```
1: Phase 1: Generate Sorted Runs
2:  $run\_count \leftarrow 0$ 
3:  $buffer \leftarrow$  empty list
4: while  $R$  has more tuples do
5:   Fill  $buffer$  with up to  $M$  tuples from  $R$ 
6:    $\text{INMEMORYSORT}(buffer, K)$ 
7:   Write  $buffer$  to temporary file  $temp\_run_{run\_count}$ 
8:    $run\_count \leftarrow run\_count + 1$ 
9:   Clear  $buffer$ 
10: end while
11: Phase 2: Merge Sorted Runs
12: Initialize priority queue  $PQ$  with first tuple from each run
13: Open output file  $R'$ 
14: while  $PQ$  is not empty do
15:    $(tuple, run\_id) \leftarrow PQ.extractMin()$ 
16:   Write  $tuple$  to  $R'$ 
17:   if  $temp\_run_{run\_id}$  has more tuples then
18:      $next\_tuple \leftarrow$  read next tuple from  $temp\_run_{run\_id}$ 
19:      $PQ.insert((next\_tuple, run\_id))$ 
20:   end if
21: end while
22: Delete all temporary run files
23: return  $R'$ 
```

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*
- *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

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

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

10 Integration with Current Implementation

10.1 Extension Points

The current architecture provides clear extension points:

Listing 11: Storage Interface Extension Point

```

1 // Current mock implementation
2 class SequentialScanNode : public PhysicalPlanNode {
3     std::vector<Tuple> mock_data; // Replace with storage
4     interface
5 public:
6     TupleBatch get_next_batch() override {
7         // Current: iterate over mock_data
8         // Future: integrate with buffer pool manager
9
10    TupleBatch batch;
11    // TODO: Replace with real storage access

```

```

12         for (size_t i = current_position; i < end_pos; ++i) {
13             if (passes_filter(mock_data[i])) {
14                 batch.add_tuple(mock_data[i]);
15             }
16         }
17         return batch;
18     }
19 };
20
21 // Future storage-integrated implementation
22 class StorageSequentialScanNode : public PhysicalPlanNode {
23     TableOid table_oid;
24     std::shared_ptr<BufferPoolManager> buffer_pool_;
25     std::shared_ptr<CatalogManager> catalog_;
26
27 public:
28     TupleBatch get_next_batch() override {
29         TupleBatch batch;
30
31         // Get table metadata from catalog
32         auto table_info = catalog_->get_table_info(table_oid);
33
34         // Iterate through pages using buffer pool
35         while (current_page_id <= table_info->last_page_id) {
36             Page* page = buffer_pool_->fetch_page(current_page_id)
37                 ;
38
39             // Extract tuples from page
40             auto page_tuples = extract_tuples_from_page(page,
41                 table_info->schema);
42
43             for (const auto& tuple : page_tuples) {
44                 if (passes_filter(tuple)) {
45                     batch.add_tuple(tuple);
46                     if (batch.is_full()) break;
47                 }
48             }
49
50             buffer_pool_->unpin_page(current_page_id, false);
51             if (batch.is_full()) break;
52
53             current_page_id++;
54         }
55
56         return batch;
57     }
58 };

```

10.2 Backward Compatibility

Extensions maintain compatibility with existing interfaces:

Listing 12: Interface Compatibility Design

```
1 // Abstract base maintains current interface
2 class PhysicalPlanNode {
3 public:
4     virtual TupleBatch get_next_batch() = 0;
5     virtual void reset() = 0;
6     // Interface remains stable
7 };
8
9 // Extensions add new capabilities
10 class StorageAwarePhysicalPlanNode : public PhysicalPlanNode {
11 protected:
12     std::shared_ptr<StorageManager> storage_manager_;
13     std::shared_ptr<TransactionManager> txn_manager_;
14
15 public:
16     // Existing interface
17     TupleBatch get_next_batch() override = 0;
18     void reset() override = 0;
19
20     // New storage-aware methods
21     virtual void set_storage_manager(std::shared_ptr<
22         StorageManager> sm) {
23         storage_manager_ = sm;
24     }
25
26     virtual void set_transaction_context(TransactionId txn_id) {
27         current_txn_id_ = txn_id;
28     }
29 };
```

11 Performance Analysis and Benchmarking

11.1 Current Performance Characteristics

The existing implementation provides a baseline for performance comparison:

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*

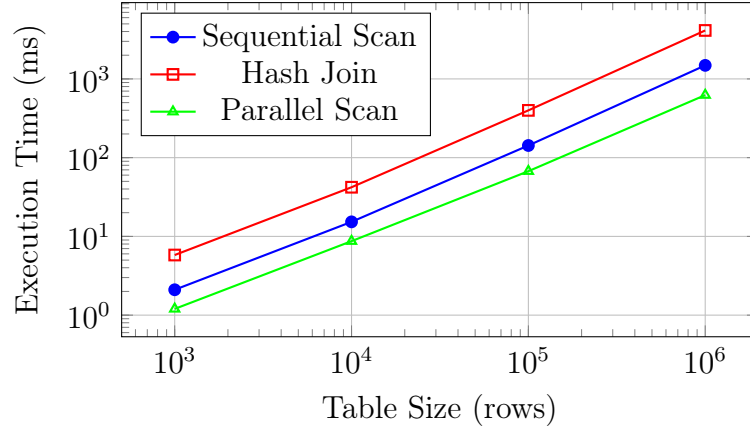


Figure 12: Performance Scaling with Mock Data

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