Building a High-Performance Mini SQLite-Based Database System

From Concept to Implementation

Database Systems Architecture

July 8, 2025

Table of Contents

Project Vision

Goal

Build a SQLite-inspired database system optimized for modern hardware constraints

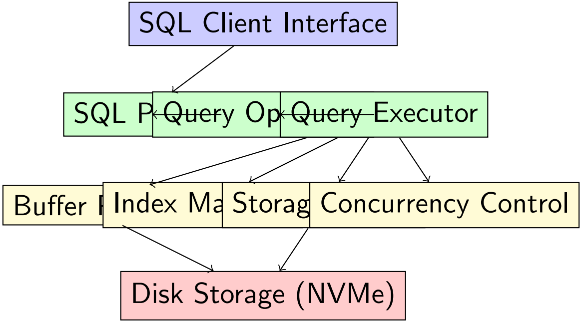
Traditional SQLite Constraints (2000s):

- Limited RAM (64MB-1GB)
- Slow disk I/O
- Single-core CPUs
- Conservative memory usage

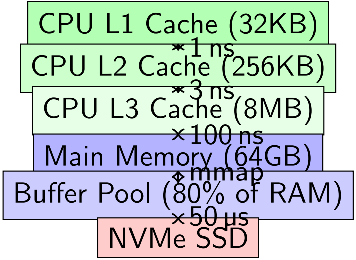
Modern Reality (2025):

- Abundant RAM (16-128GB)
- Fast NVMe SSDs
- Multi-core CPUs (8-32 cores)
- Speed is everything

System Architecture - High Level



Memory-First Architecture



Key Principles:

- Keep hot data in RAM
- Minimize disk I/O
- Optimize for cache locality
- Use memory-mapped files

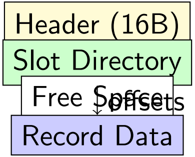
Implementation:

- 80-90% RAM for buffer pool
- Async disk writes
- Prefetching algorithms

Storage Engine - Data Structures

Listing 1: Page Structure

```
class Page {
    static const size_t PAGE_SIZE = 4096;

    struct Header {
        uint32_t page_id;
        uint16_t free_space;
        uint16_t slot_count;
        uint32_t checksum;
    };

    Header header;
    std::vector<uint16_t> slot_directory;
    std::vector<uint8_t> data;

public:
    bool insert_record(const Record&
         record);
    Record get_record(uint16_t slot_id);
```



Advantages:

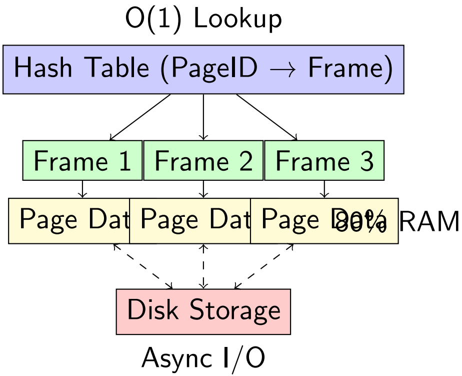
- Cache-friendly layout
- Variable-length records

Buffer Pool Manager

Listing 2: Buffer Pool Implementation

```
class BufferPool {
    struct BufferFrame {
        Page* page;
        uint32_t page_id;
        bool is_dirty;
        uint32_t pin_count;
        std::chrono::steady_clock::time_point last_access;
    };

    std::unordered_map<uint32_t, BufferFrame*> page_table;
    std::vector<BufferFrame> buffer_frames;
    std::mutex buffer_mutex;

public:
    Page* get_page(uint32_t page_id);
    void unpin_page(uint32_t page_id, bool is_dirty);
    void flush_all_pages();

private:
```

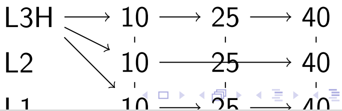
Buffer Pool - Memory Management Strategy



Skip List Index Implementation

Listing 3: Skip List Node

```
template<typename Key, typename Value>
class SkipListNode {
    Key key;
    Value value;
    std::vector<SkipListNode*> forward;

public:
    SkipListNode(Key k, Value v, int level)
        : key(k), value(v), forward(level + 1) {}

    Key get_key() const {
        return key; }
```

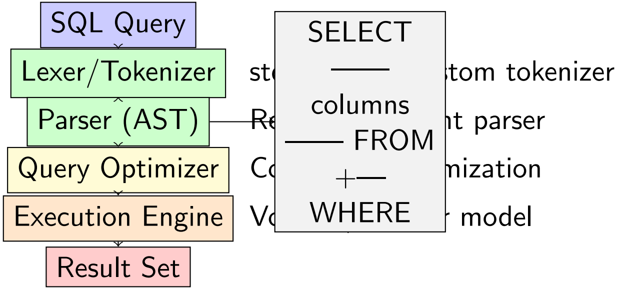


Hash Index for Equality Queries

Listing 4: Robin Hood Hash Table

```
template<typename Key, typename Value>
class RobinHoodHashTable {
    struct Entry {
        Key key;
        Value value;
        uint32_t hash;
        uint32_t distance;  // Distance from ideal position
        bool is_occupied;
    };

    std::vector<Entry> table;
    size_t capacity;
    size_t size;

public:
    bool insert(const Key& key, const Value& value);
    bool find(const Key& key, Value& value);
    bool remove(const Key& key);
```

Query Processing Flow



SQL Parser - AST Construction

Listing 5: Abstract Syntax Tree Nodes

```
class ASTNode {
public:
    virtual ~ASTNode() = default;
    virtual void accept(ASTVisitor& visitor) = 0;
};

class SelectStatement : public ASTNode {
    std::vector<std::unique_ptr<Expression>> select_list;
    std::unique_ptr<FromClause> from_clause;
    std::unique_ptr<WhereClause> where_clause;
    std::unique_ptr<OrderByClause> order_by_clause;

public:
    void accept(ASTVisitor& visitor) override {
        visitor.visit(*this);
    }
};

class TableReference : public ASTNode {
```

Query Optimizer - Cost-Based Decisions

Listing 6: Query Optimization Framework

```
class QueryOptimizer {
    struct Statistics {
        size_t table_size;
        size_t distinct_values;
        double selectivity;
        std::unordered_map<std::string, size_t> column_stats;
    };

    std::unordered_map<std::string, Statistics> table_stats;

public:
    std::unique_ptr<ExecutionPlan> optimize(const ASTNode& query);

private:
    double estimate_cost(const ExecutionPlan& plan);
    std::vector<ExecutionPlan> generate_plans(const ASTNode& query);
    ExecutionPlan select_best_plan(const std::vector<ExecutionPlan>& plans);

    // Optimization rules
```

CPU Parallelization with OpenMP

Listing 7: Parallel Table Scan

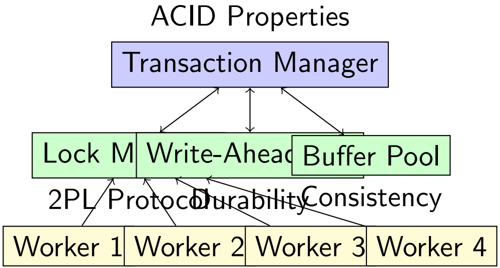
```
class TableScan {
    std::vector<Page*> pages;
    Predicate where_condition;

public:
    std::vector<Record> scan_parallel() {
        std::vector<Record> results;
        std::mutex results_mutex;

        #pragma omp parallel for
        for (size_t i = 0; i < pages.size(); ++i) {
            std::vector<Record> local_results;

            // Scan page i
            for (const auto& record : pages[i]->get_records()) {
                if (where_condition.evaluate(record)) {
                    local_results.push_back(record);
                }
            }
```

SIMD Vectorization for Batch Operations

Listing 8: Vectorized Comparison Operations

```
#include <immintrin.h>

class VectorizedOperations {
public:
    // Compare 8 integers simultaneously
    static std::vector<bool> compare_greater_than(
        const std::vector<int32_t>& values,
        int32_t threshold) {

        std::vector<bool> results(values.size());
        const __m256i threshold_vec = _mm256_set1_epi32(threshold);

        size_t i = 0;
        for (; i + 8 <= values.size(); i += 8) {
            __m256i values_vec = _mm256_loadu_si256(
                reinterpret_cast<const __m256i*>(&values[i]));

            __m256i cmp_result = _mm256_cmpgt_epi32(values_vec, threshold_vec);
```

Concurrency Control Architecture



Concurrency Strategy:

- **Two-Phase Locking (2PL)**: Strict locking protocol
- **Write-Ahead Logging**: Ensure durability and recovery

Cache-Conscious Data Structures

Listing 9: Cache-Friendly Record Layout

```
// Traditional approach - poor cache locality
struct Record_Bad {
    std::string* name;      // Pointer to heap
    int* age;               // Pointer to heap
    double* salary;         // Pointer to heap
};

// Cache-friendly approach
struct Record_Good {
    char name[64];          // Fixed-size, inline storage
    int age;                // Value stored inline
    double salary;          // Value stored inline

    // Padding to align to cache line boundary
    char padding[64 - sizeof(name) - sizeof(age) - sizeof(salary)];
} __attribute__((aligned(64)));

// Columnar storage for analytical queries
```

Async I/O and Prefetching

Listing 10: Asynchronous Disk Operations

```
#include <future>
#include <thread>

class AsyncDiskManager {
    std::thread io_thread;
    std::queue<std::packaged_task<void()>> io_queue;
    std::mutex queue_mutex;
    std::condition_variable cv;

public:
    AsyncDiskManager() : io_thread(&AsyncDiskManager::io_worker, this) {}

    std::future<Page*> read_page_async(uint32_t page_id) {
        auto task = std::packaged_task<Page*()>(
            [this, page_id]() { return read_page_from_disk(page_id); });

        auto future = task.get_future();

        {
```

Memory Hierarchy Optimization

L1 Cache: 32KB, 1ns

L2 Cache: 256KB, 3ns

L3 Cache: 8MB, 10ns

RAM: 64GB, 100ns

NVMe SSD:50 µs

Critical Path

Optimize

Minimize

- Prefetch next cache line
- Minimize random access
- Batch similar operations
- Use SIMD instructions
- Compress data when possible
- Keep hot data in cache
- Memory-mapped files
- Async I/O operations
- Batch disk writes

Development Phases

Phase 1: Foundation

- Basic storage engine
- Simple B+ tree index
- Basic SQL parser

↓

Phase 2: Core Features

- Transaction support
- Concurrency control

↓

Phase 3: Optimization

- Query optimization
- Parallel execution
- SIMD operations
- Cache optimization

↓

Phase 4: Advanced

- Advanced indexing
- Compression
- Distributed features

Phase 1: Foundation Implementation

Listing 11: Basic Storage Engine Skeleton

```
// Core classes to implement first
class StorageEngine {
    std::unique_ptr<BufferPool> buffer_pool;
    std::unique_ptr<DiskManager> disk_manager;

public:
    bool create_table(const std::string& table_name,
                      const TableSchema& schema);
    bool insert_record(const std::string& table_name,
                       const Record& record);
    std::vector<Record> scan_table(const std::string& table_name);
};

class DiskManager {
    std::fstream db_file;
    std::mutex file_mutex;

public:
    Page* read_page(uint32_t page_id);
```

Phase 2: Adding SQL and Transactions

Listing 12: SQL Parser and Transaction Manager

```
// SQL Parser using recursive descent
class SQLParser {
    std::regex select_pattern;
    std::regex insert_pattern;
    std::regex create_table_pattern;

public:
    SQLParser() {
        select_pattern = std::regex(R"(SELECT\s+(.+)\s+FROM\s+(\w+)(?:\s+WHERE\s+(.+))?)
            ");
        insert_pattern = std::regex(R"(INSERT\s+INTO\s+(\w+)\s+VALUES\s*\((.+)\))");
    }

    std::unique_ptr<Statement> parse(const std::string& sql);
};

// Transaction Manager
class TransactionManager {
    std::atomic<uint64_t> next_txn_id{1};
```

Phase 3: Performance Optimization

Listing 13: Parallel Query Execution

```
// Thread pool for parallel execution
class ThreadPool {
    std::vector<std::thread> workers;
    std::queue<std::function<void()>> tasks;
    std::mutex queue_mutex;
    std::condition_variable condition;
    bool stop = false;

public:
    ThreadPool(size_t num_threads = std::thread::hardware_concurrency());

    template<typename F>
    auto enqueue(F&& f) -> std::future<decltype(f())> {
        auto task = std::make_shared<std::packaged_task<decltype(f())()>>(
            std::forward<F>(f));

        auto future = task->get_future();
        {
            std::unique_lock<std::mutex> lock(queue_mutex);
```

Phase 4: Advanced Features

Listing 14: Compression and Advanced Indexing

```
// Data compression for storage efficiency
class CompressionManager {
public:
    // Dictionary compression for strings
    std::vector<uint8_t> compress_string_column(
        const std::vector<std::string>& strings) {
        std::unordered_map<std::string, uint32_t> dictionary;
        std::vector<uint32_t> compressed_data;

        uint32_t next_id = 0;
        for (const auto& str : strings) {
            if (dictionary.find(str) == dictionary.end()) {
                dictionary[str] = next_id++;
            }
            compressed_data.push_back(dictionary[str]);
        }

        return serialize_compressed_data(dictionary, compressed_data);
    }
```

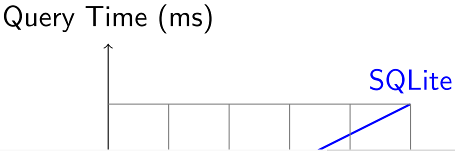
Performance Testing Framework

Listing 15: Benchmark Suite Implementation

```
#include <chrono>
#include <random>

class DatabaseBenchmark {
    std::unique_ptr<StorageEngine> engine;
    std::mt19937 rng;

public:
    // TPC-C style benchmark
    void benchmark_oltp_workload() {
        const size_t num_transactions = 10000;
        const size_t num_threads = std::thread::hardware_concurrency();

        auto start = std::chrono::high_resolution_clock::now();

        std::vector<std::thread> workers;
        for (size_t i = 0; i < num_threads; ++i) {
            workers.emplace_back([this, num_transactions, num_threads, i]() {
                for (size_t j = i; j < num_transactions; j += num_threads) {
```

Performance Metrics & Targets

Operation	Target	Measurement	Comparison
Point Query	¡ 0.1ms	std::chrono	vs SQLite
Range Scan	1M rows/sec	Throughput	vs PostgreSQL
Insert	10K TPS	Transactions/sec	vs MySQL
Parallel Scan	4x speedup	vs serial	CPU cores
Memory Usage	80% buffer hit	Cache ratio	vs traditional



Key Takeaways

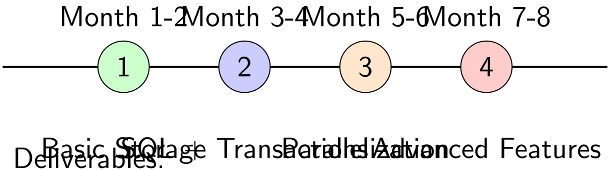
Architecture Principles

- **Memory-first design**: Leverage abundant RAM for speed
- **CPU parallelization**: Use all available cores with OpenMP
- **Cache-conscious data structures**: Optimize for modern CPU hierarchy
- **Async I/O**: Overlap computation and disk operations

Implementation Strategy

- **Start simple**: Basic storage engine with file I/O
- **Iterate quickly**: Add features incrementally
- **Measure everything**: Benchmark each optimization

Development Timeline



Deliverables:

- Working prototype
- Performance benchmarks
- Documentation
- Test suite

Questions & Discussion

Thank You!

Contact & Resources

- **Source code**: Will be available on GitHub
- **Documentation**: Comprehensive API documentation
- **Benchmarks**: Performance comparison results
- **Community**: Open source contribution guidelines