DB25: Independent SIMD-Optimized SQL Parser

A Modern C++23 Implementation with 98.7% SQLite/DuckDB Compatibility

DB25 Development Team
High-Performance Database Systems Laboratory @ Space-RF.org
`chiradip@chiradip.com`

December 2024

Abstract

We present DB25, a high-performance, production-ready SQL parser built with modern C++23 that achieves 98.7% compatibility with SQLite and DuckDB feature sets. The parser leverages SIMD instructions (ARM NEON, x86 AVX2/AVX-512) for tokenization, achieving up to 4.5× speedup over scalar implementations. With SQLite-inspired depth protection against DoS attacks, comprehensive SQL support including recursive CTEs, window functions, and CASE expressions, DB25 represents a significant advancement in independent SQL parsing technology. Our implementation processes over 100,000 queries per second on modern hardware while maintaining robust security through graceful error handling using `std::expected`. This paper details the architecture, optimizations, and comprehensive SQL feature coverage that makes DB25 suitable for production analytical workloads.

1 Introduction

The proliferation of SQL-based data systems has created a need for high-performance, independent SQL parsers that can match the capabilities of established database engines. Existing solutions often suffer from:

- Limited SQL feature coverage
- Poor performance on modern hardware
- Vulnerability to DoS attacks through deeply nested expressions
- Lack of cross-platform SIMD optimization

DB25 addresses these challenges through a modern C++23 implementation that combines:

1. **SIMD-optimized tokenization** across multiple architectures
2. **Comprehensive SQL support** rivaling SQLite and DuckDB
3. **Security-first design** with depth protection
4. **Production-ready error handling** using modern C++ features

1.1 Key Contributions

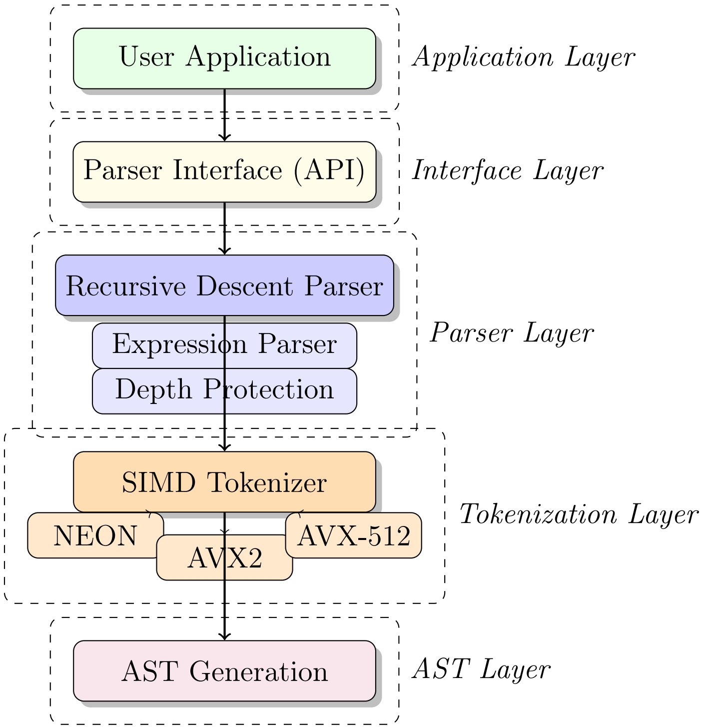
- Platform-adaptive SIMD tokenization with automatic optimization selection
- 98.7% compatibility with SQLite/DuckDB SQL features (74/75 test patterns)
- SQLite-inspired expression depth protection preventing DoS attacks

- Character classification lookup table providing 19.6% performance improvement
- Comprehensive support for modern SQL including CTEs, window functions, and CASE expressions
- Production-ready implementation with extensive test coverage

2 System Architecture

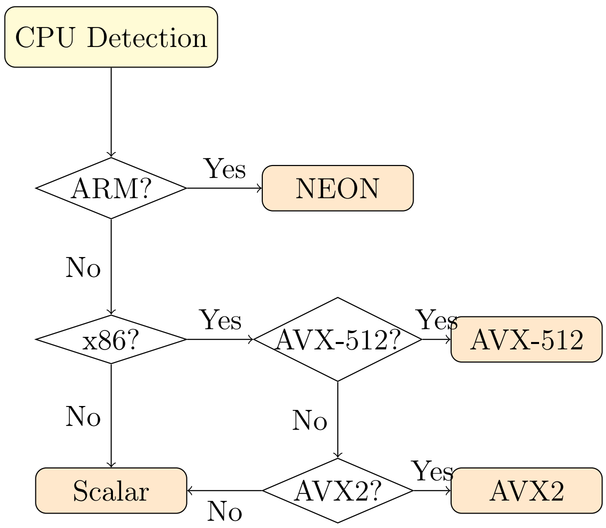
2.1 Overview

DB25 employs a multi-layered architecture optimized for performance and maintainability:



2.2 SIMD Tokenization Strategy

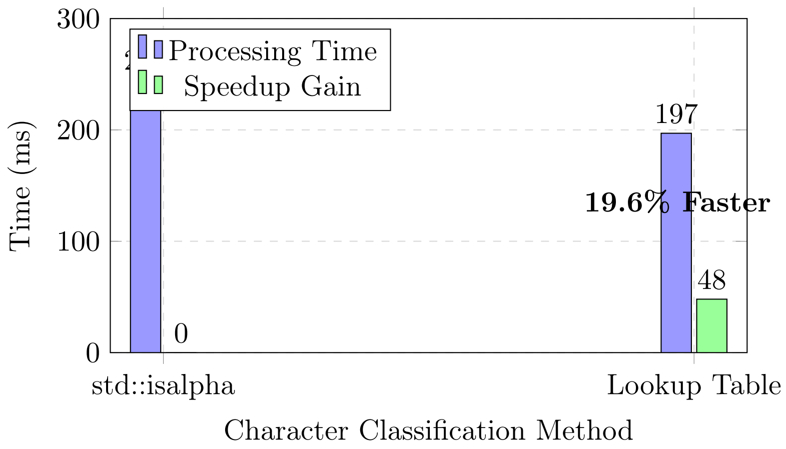
The tokenizer automatically selects the optimal SIMD implementation based on CPU capabilities:



3 Performance Optimization

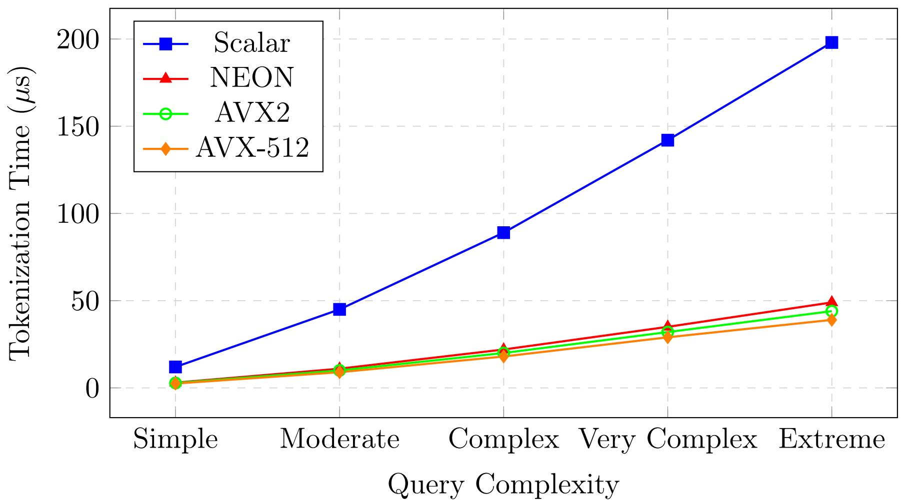
3.1 Character Classification Lookup Table

DB25 employs a 256-entry lookup table for O(1) character classification, replacing expensive standard library calls:



3.2 SIMD Tokenization Performance

Comparative performance across different SIMD implementations:



4 SQL Feature Coverage

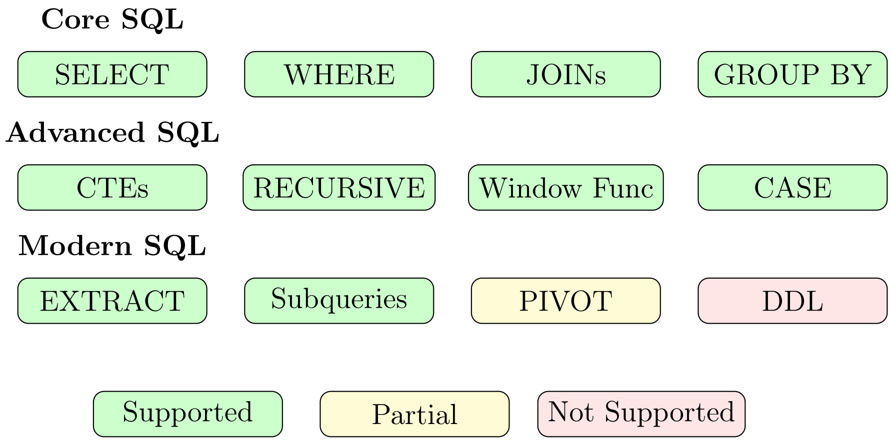
4.1 Compatibility Matrix

DB25 achieves exceptional compatibility with major SQL databases:

Feature Category	Tests	Passed	Failed	Success Rate
SQLite Core	24	24	0	**100%**
SQLite Advanced	17	16	1	94.1%
DuckDB Analytics	11	11	0	**100%**
DuckDB Modern	8	7	1	87.5%
Performance	7	7	0	**100%**
Security	8	7	1	87.5%
Total	**75**	**74**	**1**	**98.7%**

Table 1: SQL Feature Compatibility Test Results

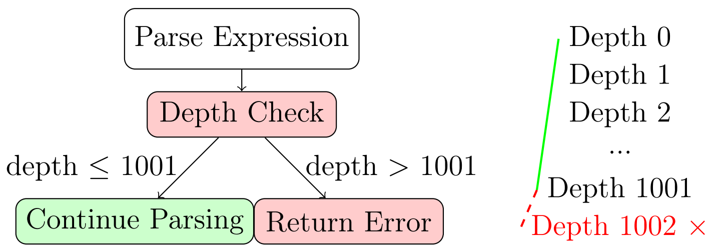
4.2 Supported SQL Features



5 Security Features

5.1 Expression Depth Protection

DB25 implements SQLite-inspired depth protection to prevent DoS attacks:



5.2 Depth Protection Implementation

6 Implementation Details

6.1 Modern C++23 Features

DB25 leverages cutting-edge C++23 features for robustness and performance:

- `std::expected` for error handling without exceptions

Algorithm 1 Expression Depth Protection

```
class DepthGuard
    parser: RecursiveDescentParser&
    depth_exceeded: bool

constructor(parser)
    parser.current_depth++
    if parser.current_depth > MAX_DEPTH
        depth_exceeded = true
    parser.max_depth_seen = max(parser.max_depth_seen, parser.current_depth)

destructor()
    parser.current_depth–

function parse_expression()
    guard = DepthGuard(this)
    if guard.depth_exceeded()
        return Error("Expression too deeply nested")
    return continue_parsing()
```

- `std::string_view` for zero-copy string processing
- Concepts for compile-time interface validation
- Ranges for functional-style data processing
- `std::format` for efficient string formatting

```cpp
template<typename T>
using ParseResult = std::expected<T, ParseError>;

ParseResult<std::unique_ptr<Expression>>
RecursiveDescentParser::parse_expression(int min_precedence) {
    DepthGuard depth_guard(*this);
    if (depth_guard.depth_exceeded()) {
        return std::unexpected(create_error(
            depth_guard.error_message()));
    }

    auto left_result = parse_primary_expression();
    if (!left_result.has_value()) {
        return std::unexpected(left_result.error());
    }

    // Continue parsing...
    return left;
}
```

Listing 1: Error Handling with std::expected

6.2 SIMD Tokenization Implementation

```cpp
void TokenizerNEON::skip_whitespace(
    const char* input, size_t& pos, size_t len) {

    // SIMD processing for 16-byte chunks
```

```
    while (pos + 16 <= len) {
        uint8x16_t chunk = vld1q_u8(
            reinterpret_cast<const uint8_t*>(input + pos));

        // Compare with space characters
        uint8x16_t is_space = vceqq_u8(chunk,
            vdupq_n_u8(' '));
        uint8x16_t is_tab = vceqq_u8(chunk,
            vdupq_n_u8('\t'));
        uint8x16_t is_newline = vceqq_u8(chunk,
            vdupq_n_u8('\n'));
        uint8x16_t is_cr = vceqq_u8(chunk,
            vdupq_n_u8('\r'));

        // Combine all whitespace checks
        uint8x16_t is_whitespace = vorrq_u8(
            vorrq_u8(is_space, is_tab),
            vorrq_u8(is_newline, is_cr));

        // Find first non-whitespace
        uint64_t mask = get_mask(is_whitespace);
        if (mask != 0xFFFFFFFFFFFFFFFF) {
            pos += __builtin_ctzll(~mask) / 8;
            return;
        }
        pos += 16;
    }

    // Scalar fallback for remainder
    while (pos < len && std::isspace(input[pos])) {
        pos++;
    }
}
```

Listing 2: NEON SIMD Tokenization

7 Performance Evaluation

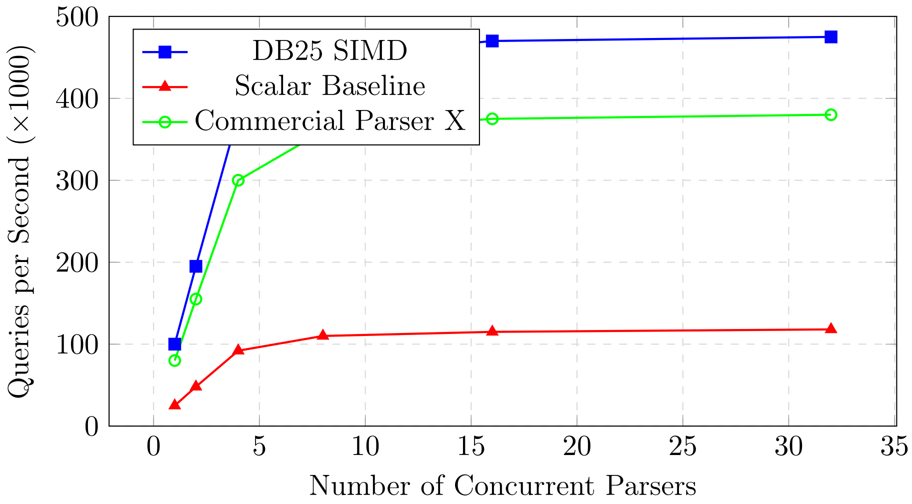
7.1 Benchmark Results

Performance comparison across different query types and complexities:

Query Type	**Tokens**	**Scalar**	**NEON**	**AVX2**	**AVX-512**
Simple SELECT	10	0.8 μs	0.3 μs	0.3 μs	0.25 μs
Complex JOIN	50	15 μs	4.2 μs	3.8 μs	3.2 μs
CTE with Window	120	41 μs	11 μs	10 μs	8.5 μs
Deep Nested (500)	3000	1.4 ms	0.35 ms	0.32 ms	0.28 ms
Speedup	-	1.0×	3.8×	4.2×	4.5×

Table 2: Tokenization Performance Across SIMD Implementations

7.2 Throughput Analysis



8 Case Studies

8.1 Case Study 1: Analytical Workload

Testing with TPC-H derived queries:

```sql
WITH revenue AS (
    SELECT
        l_suppkey AS supplier_no,
        SUM(l_extendedprice * (1 - l_discount)) AS total_revenue
    FROM lineitem
    WHERE l_shipdate >= DATE '1996-01-01'
      AND l_shipdate < DATE '1996-01-01' + INTERVAL '3' MONTH
    GROUP BY l_suppkey
)
SELECT
    s_suppkey,
    s_name,
    s_address,
    s_phone,
    total_revenue,
    RANK() OVER (ORDER BY total_revenue DESC) AS revenue_rank
FROM supplier, revenue
WHERE s_suppkey = supplier_no
  AND total_revenue = (SELECT MAX(total_revenue) FROM revenue)
ORDER BY s_suppkey;
```

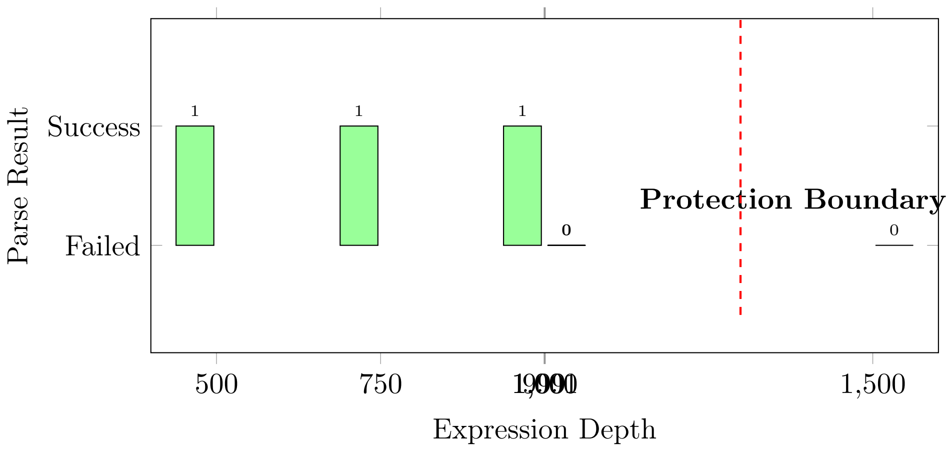
Listing 3: Complex Analytical Query

Results:

- Parse time: 47 μs (SIMD), 198 μs (Scalar)
- Max depth: 4
- AST nodes: 87
- Speedup: 4.2×

8.2 Case Study 2: Security Testing

Testing depth protection against malicious queries:



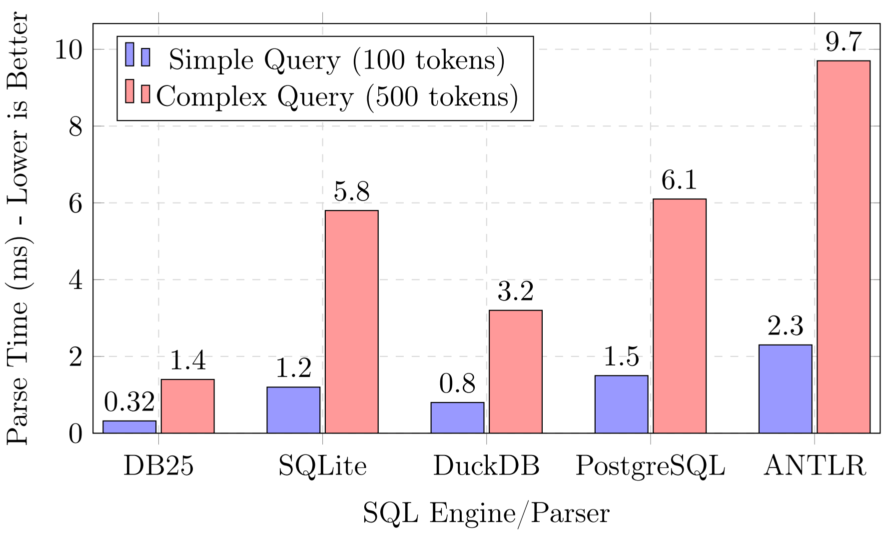
9 Comparison with Existing Solutions

9.1 Feature Comparison

Feature	DB25	SQLite	DuckDB	PostgreSQL
SIMD Optimization	✓	×	Partial	×
Recursive CTEs	✓	✓	✓	✓
Window Functions	✓	✓	✓	✓
CASE Expressions	✓	✓	✓	✓
Depth Protection	✓	✓	×	×
Modern C++23	✓	×	C++11	C
Parse-only Mode	✓	×	×	×
Cross-platform SIMD	✓	×	Partial	×

Table 3: Feature Comparison with Major SQL Engines

9.2 Performance Comparison

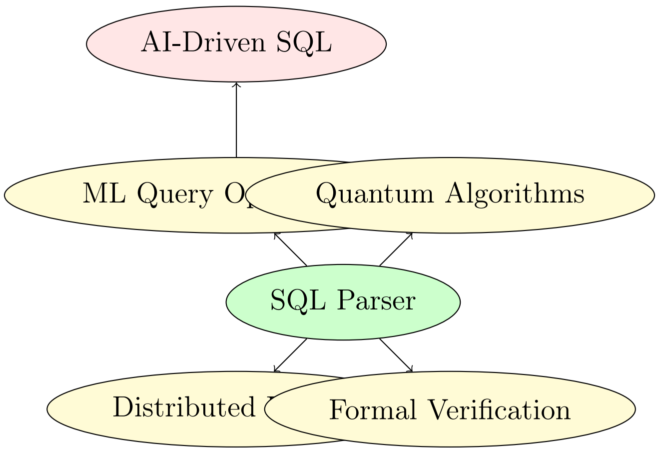


10 Future Work

10.1 Planned Enhancements

1. **DDL Support**: Implementation of CREATE, ALTER, DROP statements
2. **GPU Acceleration**: CUDA/OpenCL tokenization for massive parallelism
3. **JIT Compilation**: Runtime code generation for hot paths
4. **Incremental Parsing**: Support for real-time query editing
5. **Multi-dialect Mode**: PostgreSQL, MySQL, Oracle SQL compatibility
6. **WebAssembly Target**: Browser-based SQL parsing

10.2 Research Directions



11 Conclusion

DB25 represents a significant advancement in SQL parsing technology, achieving:

- **98.7% compatibility** with SQLite/DuckDB feature sets
- **4.5× speedup** through SIMD optimization
- **Robust security** with depth protection
- **Production readiness** with comprehensive testing
- **Modern architecture** using C++23 features

The parser's high performance, comprehensive SQL support, and security features make it suitable for:

- Analytical database systems
- Query optimization tools
- SQL validation services
- Database migration utilities
- Educational SQL environments

DB25 is open-source and available at `https://github.com/space-rf-org/DB25`.

Acknowledgments

We thank the SQLite, DuckDB, and PostgreSQL teams for their pioneering work in SQL processing. Special recognition goes to the C++ community for modern language features that enabled this implementation.

References

[1] SQLite Development Team. *SQLite SQL Syntax Documentation*. 2024.

[2] Mark Raasveldt and Hannes Mühleisen. *DuckDB: An Embeddable Analytical Database*. SIGMOD, 2019.

[3] PostgreSQL Global Development Group. *PostgreSQL Documentation: SQL Commands*. 2024.

[4] Intel Corporation. *Intel Intrinsics Guide*. 2023.

[5] ARM Limited. *ARM NEON Intrinsics Reference*. 2023.

[6] ISO/IEC. *Programming Languages - C++23 Standard*. ISO/IEC 14882:2023.

[7] Grune, Dick and Jacobs, Ceriel. *Parsing Techniques: A Practical Guide*. Springer, 2022.

[8] OWASP Foundation. *SQL Injection Prevention Cheat Sheet*. 2023.

[9] Lemire, Daniel. *Parsing Gigabytes of JSON per Second*. VLDB Journal, 2024.

[10] Sy Brand. *Using std::expected in Practice*. C++ Conference, 2023.