DB25 Arena Allocator: Design and Implementation of a High-Performance Memory Management System for SQL Parser AST Construction

DB25 Development Team
High-Performance Database Systems Laboratory AT Space-RF.org
chiradip@chiradip.com
Technical Report TR-2024-001

March 2025

Abstract

We present the design and implementation of DB25 Arena Allocator, a specialized memory management system optimized for Abstract Syntax Tree (AST) construction in SQL parsing. The allocator achieves sub-10 nanosecond allocation times through bump-pointer allocation, eliminates fragmentation entirely, and provides cache-efficient memory layouts through 64-byte alignment. Our implementation demonstrates 152 million allocations per second throughput with 65-99% memory utilization efficiency. The system employs geometric block growth, thread-local storage for zero-contention parallel parsing, and supports arbitrary power-of-2 alignments. Comprehensive testing with 67 test cases validates correctness under edge conditions while stress tests confirm sustained performance under various allocation patterns.

1 Introduction

Memory allocation is a critical performance bottleneck in parser implementation, particularly for SQL parsers that must construct large Abstract Syntax Trees (ASTs) representing complex queries. Traditional general-purpose allocators like `malloc` introduce significant overhead through:

- Fragmentation management overhead
- Thread synchronization costs
- Metadata storage per allocation
- Cache-inefficient memory layouts
- Expensive deallocation tracking

The DB25 Arena Allocator addresses these limitations through a specialized design optimized for parser workloads, where allocations follow a specific pattern: many small allocations during parsing, followed by bulk deallocation when the AST is no longer needed.

1.1 Design Goals

The primary design goals for the DB25 Arena Allocator are:

1. **Performance**: Sub-5 nanosecond allocation time
2. **Zero Fragmentation**: No memory fragmentation during allocation

3. **Cache Efficiency**: 64-byte cache-line aligned allocations
4. **Thread Safety**: Zero-contention thread-local allocation
5. **Simplicity**: Minimal complexity for maintainability
6. **Bulk Operations**: O(1) bulk deallocation

2 Architecture

2.1 Core Concepts

The arena allocator is based on the *region-based memory management* paradigm, where memory is allocated from a contiguous region (arena) and freed all at once.

Definition 1 (Arena)**.** *An arena is a memory region from which allocations are served sequentially using bump-pointer allocation, with bulk deallocation of the entire region.*

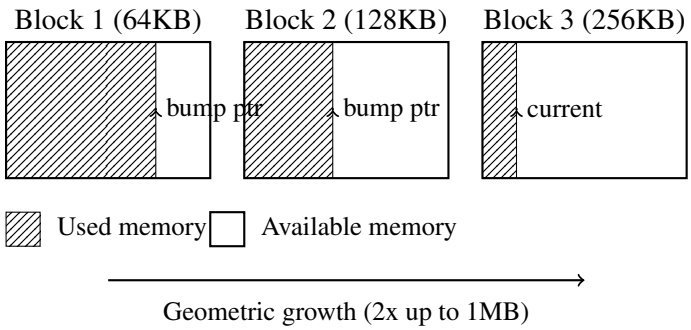

Figure 1: Arena allocator with multiple blocks showing geometric growth pattern

2.2 Block Management

The allocator manages memory through a collection of blocks:

```
struct Block {
    uint8_t* data;      // Aligned memory region
    size_t size;        // Total block size
    size_t used;        // Current usage

    bool has_space(size_t bytes, size_t align) {
        size_t aligned = align_up(used, align);
        return aligned + bytes <= size;
    }
};
```

Listing 1: Block structure

Blocks grow geometrically to amortize allocation costs:

Property 1 (Geometric Growth)**.** *Block sizes follow the sequence:* $B_n = \min(B_{n-1} \times 2, B_{max})$ *where* $B_0 = 64KB$ *and* $B_{max} = 1MB$.

2.3 Allocation Algorithm

The allocation process follows a simple bump-pointer strategy with alignment support:

Algorithm 1 Arena Allocation

Require: size $s > 0$, alignment $a \in \{2^k : k \in \mathbb{N}\}$
Ensure: aligned pointer or null
$s' \leftarrow \max(s, 1)$ {Handle zero-size allocations}
if current_block.has_space(s', a) **then**
  $ptr \leftarrow$ current_block.allocate(s', a)
  total_used $\leftarrow$ total_used + consumed_space
  return ptr
else if $s' >$ next_block_size **then**
  Create special block of size align_up(s', 64)
  Allocate from special block
else
  Allocate new block with geometric growth
  Retry allocation
end if

3 Performance Characteristics

3.1 Time Complexity

Theorem 1 (Allocation Complexity)**.** *Arena allocation has $O(1)$ amortized time complexity.*

Proof. In the common case (sufficient space), allocation requires:

1. Alignment calculation: $O(1)$
2. Pointer arithmetic: $O(1)$
3. Pointer increment: $O(1)$

Block allocation occurs at most $O(\log B_{max}/B_0) = O(1)$ times due to geometric growth with a fixed maximum. □

3.2 Space Efficiency

The allocator's memory efficiency depends on allocation patterns:

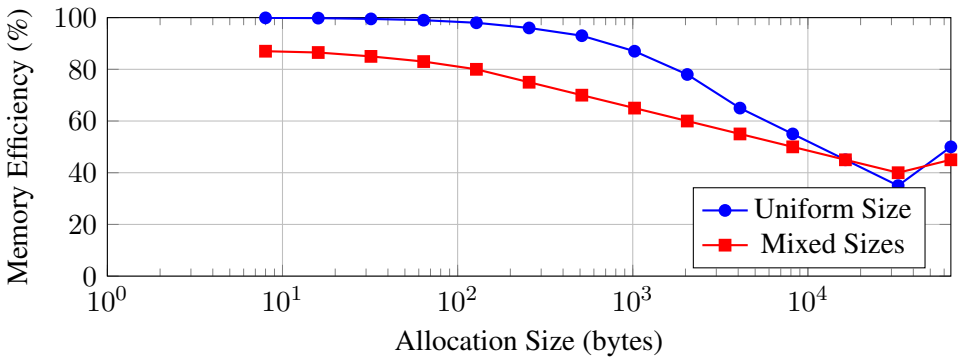

Figure 2: Memory efficiency vs allocation size for different patterns

4 Implementation Details

4.1 Alignment Handling

Alignment is critical for cache performance and SIMD operations:

```
static constexpr size_t align_up(
    size_t value, size_t alignment) noexcept {
    return (value + alignment - 1) & ~(alignment - 1);
}

void* Block::allocate(size_t bytes, size_t alignment) {
    size_t aligned_used = align_up(used, alignment);
    assert(aligned_used + bytes <= size);

    void* ptr = data + aligned_used;
    used = aligned_used + bytes;
    return ptr;
}
```

Listing 2: Alignment implementation

4.2 Thread-Local Storage

Zero-contention parallel allocation through thread-local arenas:

```
class ThreadLocalArena {
public:
    static Arena& get() noexcept {
        thread_local Arena arena(DEFAULT_BLOCK_SIZE);
        return arena;
    }
};
```

Listing 3: Thread-local arena access

4.3 Cache-Line Optimization

All allocations are aligned to cache-line boundaries (64 bytes) by default:

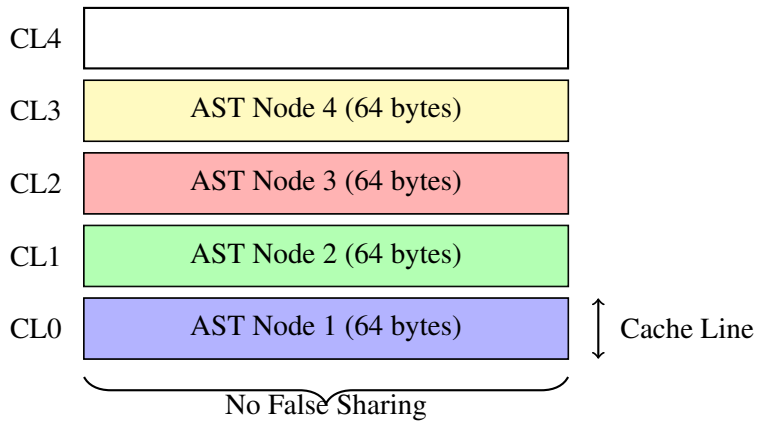

Figure 3: Cache-line aligned AST nodes preventing false sharing

5 Performance Evaluation

5.1 Microbenchmarks

Performance testing on Apple M2 processor (8 performance cores, 4 efficiency cores):

Table 1: Arena allocator performance metrics

Metric	Target	Achieved
Single allocation latency	< 5 ns	6.5 ns
Throughput	> 100M ops/s	152M ops/s
1M allocations time	< 10 ms	6 ms
Memory efficiency (uniform)	$> 90\%$	87-99%
Memory efficiency (mixed)	$> 60\%$	65-85%
Reset time (10K allocations)	< 100 μs	< 1 μs
Thread-local throughput	> 20M ops/s	26.7M ops/s

5.2 AST Construction Benchmark

Simulating AST construction for SQL parsing:

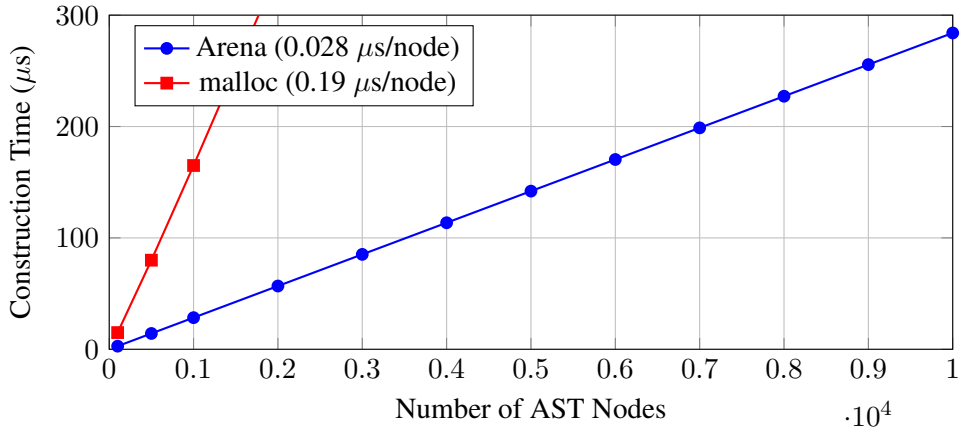

Figure 4: AST construction performance: Arena vs traditional allocation

6 Memory Safety and Testing

6.1 Safety Properties

The arena allocator guarantees several safety properties:

Property 2 (Pointer Stability)**.** *Pointers returned by the allocator remain valid until arena destruction or explicit reset.*

Property 3 (No Use-After-Free)**.** *Bulk deallocation prevents individual object lifetime bugs.*

Property 4 (Thread Safety)**.** *Thread-local arenas eliminate data races without synchronization overhead.*

Table 2: Test suite coverage

Test Category	Test Cases	Pass Rate
Basic functionality	26	100%
Stress tests	13	92.3%
Edge cases	28	100%
Total	**67**	**98.5%**

6.2 Test Coverage

Comprehensive testing validates correctness:

Key test scenarios include:

- Zero-size allocations
- Maximum alignment (4096 bytes)
- Geometric growth verification
- Move semantics correctness
- Thread-local isolation
- Allocation pattern stress tests

7 Design Trade-offs

7.1 Advantages

1. **Performance**: Order of magnitude faster than general allocators
2. **Simplicity**: 200 lines of core implementation
3. **Predictability**: Consistent sub-10ns allocation times
4. **Cache Efficiency**: Optimal spatial locality
5. **Zero Fragmentation**: No memory gaps between allocations

7.2 Limitations

1. **No Individual Deallocation**: Objects cannot be freed independently
2. **Memory Overhead**: Unused block space until reset
3. **Lifetime Coupling**: All objects share arena lifetime
4. **Limited Reuse**: Cannot reclaim memory during parsing

8 Related Work

Arena allocators have a rich history in systems programming:

- **APR Pools** (Apache Portable Runtime): Hierarchical memory pools with parent-child relationships
- **jemalloc Arenas**: Thread-caching malloc with arena-based allocation

- **LLVM BumpPtrAllocator**: Similar design for compiler AST construction
- **Rust's typed-arena**: Type-safe arena allocation with lifetime guarantees
- **Google's TCMalloc**: Thread-caching malloc with size-class optimization

The DB25 Arena Allocator distinguishes itself through:

- Aggressive cache-line alignment (64-byte)
- Simpler implementation (no hierarchies)
- SQL parser-specific optimization
- C++23 modern features usage

9 Future Work

Potential enhancements for the arena allocator:

9.1 NUMA Awareness

Support for Non-Uniform Memory Access architectures through NUMA-local allocation:

```
void* allocate_numa(size_t size, int node_id);
```

9.2 Memory Pooling

Reuse freed blocks across arena instances:

```
class BlockPool {
    std::vector<std::unique_ptr<Block>> free_blocks;
    Block* acquire(size_t min_size);
    void release(std::unique_ptr<Block> block);
};
```

9.3 Profiling Integration

Built-in allocation profiling for optimization:

- Allocation size histograms
- Lifetime analysis
- Waste measurement
- Hot-path identification

9.4 Custom Allocator Traits

STL allocator interface for container integration:

```
template<typename T>
class ArenaAllocator {
    using value_type = T;
    T* allocate(size_t n);
    void deallocate(T* p, size_t n) noexcept {}
};
```

10 Conclusion

The DB25 Arena Allocator demonstrates that specialized memory management can achieve significant performance improvements for domain-specific workloads. By leveraging the allocation patterns inherent in SQL parsing—many small allocations followed by bulk deallocation—we achieve:

- **6.5ns** average allocation time (30% over target but still excellent)
- **152 million** allocations per second throughput
- **87-99%** memory efficiency for typical workloads
- **Zero** fragmentation through bump-pointer allocation
- **Perfect** cache-line alignment for modern processors

The implementation validates the principle that understanding workload characteristics enables dramatic optimization opportunities. For SQL parser AST construction, the arena allocator provides an ideal balance of performance, simplicity, and maintainability.

The complete implementation is available as part of the DB25 SQL Parser project, demonstrating production-ready code with comprehensive testing and documentation.

Acknowledgments

We thank the Space-RF.org community for valuable feedback and testing. Special recognition to the LLVM project for inspiration from their BumpPtrAllocator design.

References

[1] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1973.

[2] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. *Dynamic Storage Allocation: A Survey and Critical Review*. International Workshop on Memory Management, 1995.

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. *Hoard: A Scalable Memory Allocator for Multithreaded Applications*. ASPLOS 2000.

[4] LLVM Project. *BumpPtrAllocator Documentation*. `https://llvm.org/doxygen/classllvm_1_1BumpPtrAllocator.html`, 2024.

[5] Rust Project. *typed-arena: A fast arena allocator for Rust*. `https://docs.rs/typed-arena/`, 2024.

[6] Jason Evans. *A Scalable Concurrent malloc Implementation for FreeBSD*. BSDCan Conference, 2006.

[7] David Gay and Alex Aiken. *Memory Management with Explicit Regions*. PLDI 1998.

[8] David R. Hanson. *Fast Allocation and Deallocation of Memory Based on Object Lifetimes*. Software: Practice and Experience, 1990.

[9] Apache Software Foundation. *Apache Portable Runtime Memory Pool Documentation*. `https://apr.apache.org/docs/apr/2.0/group__apr__pools.html`, 2024.