Breaking the `log(n)` Barrier: The SBSA Model for Constant-Time Scheduling and Storage

Aaron Cattell

June 18, 2025

Abstract

We present SBSA (Size-Based Slot Allocation), a new model that breaks through the traditional $\log(n)$ barrier in scheduling and storage. Most priority queues, B-trees, and file indexing methods rely on comparison-based logic with inherent logarithmic performance. SBSA replaces this with a spatial addressing model that enables constant-time ($O(1)$) operations through three logical coordinates: slot, thickness, and width. This paper formalizes the SBSA method, presents benchmarks against Python's `heapq`, details its application in file systems, task managers, and quantum computing workflows, and offers a new design pattern for large-scale systems. We argue that SBSA is a shift in logic organization, not just a performance trick.

1 Introduction

Comparison-based structures dominate computing: B-trees for storage, heaps for queues, and AVL trees for indexing. They all rely on $\log(n)$ comparisons, which create scaling bottlenecks. But what if every task, file, or event had a fixed, deterministic location based on pure logic?

2 The SBSA Coordinate Model

SBSA assigns each entity a unique logical address using a 3D tuple:

$$(slot, thickness, width)$$

- **Slot (s)** — a categorical class (e.g. file size, task priority)
- **Thickness (t)** — a stackable layer (e.g. thread, queue)
- **Width (w)** — a continuous scalar (e.g. duration, time)

Each triple uniquely maps to a deterministic storage path or scheduling position. No comparisons, sorting, or balancing are needed.

3 Mathematical Justification

Assuming:

- A bounded slot function $S : C \to \mathbb{Z}$
- A layer function $T : Q \to \mathbb{Z}_{\geq 0}$
- A scalar field $W \in \mathbb{R}^+$

Then the addressing space A is:

$$A = S \times T \times W$$

Each insert is a write to:

$$\texttt{storage/slot}_{S(c)}\texttt{/layer}_{T(q)}\texttt{/file}_{w}.ext$$

This guarantees $O(1)$ computation time per operation since no lookup, sorting, or rebalancing occurs.

4 Python Implementation (Storage)

sbsa_storage.py

```
class SBSAStorage:
    def __init__(self, size_classes):
        self.slots = {cls: i for i, cls in enumerate(size_classes)}

    def get_path(self, size_class, thickness, width):
        slot = self.slots[size_class]
        return f"storage/slot_{slot}/layer_{thickness}/file_{width}.txt"

    def write(self, size_class, thickness, width, content):
        path = self.get_path(size_class, thickness, width)
        os.makedirs(os.path.dirname(path), exist_ok=True)
        with open(path, 'w') as f:
            f.write(content)
```

5 Task Manager Benchmark

A benchmark against `heapq` using 10,000 tasks:

```
for count in task_counts:
    # SBSA
    for t in tasks:
        sbsa.write(*t, "task payload")
    # Heap
    heap = []
    for t in tasks:
        heapq.heappush(heap, (t[2], t))
```

Results Table

Task Count	Heap Time (s)	SBSA Time (s)
1000	0.010	0.0009
5000	0.050	0.0032
10000	0.080	0.0051

6 Quantum Queue Use Case

In a Qiskit-based demo:

```
sbsa.write_circuit("High", 0, 3.5, circuit)
    storage/slot_2/layer_0/file_3.5.qasm
```

This enables real-time circuit queuing with zero overhead indexing.

7 Collision and Determinism

SBSA guarantees:

- No runtime collisions (address fully determined by keys)
- No need for heap ordering or conflict resolution
- Consistent writes/readbacks via logical mapping

8 Conclusion

SBSA is a structural alternative to $\log(n)$ algorithms. Instead of optimizing comparison, it avoids it entirely by mapping logic into space. The result is faster, simpler, and radically scalable. It enables new thinking in schedulers, filesystems, and hybrid quantum-classical runtimes.

Historical Context

Invented by Aaron Cattell in 2024, SBSA emerged as a response to years of frustration with scaling data structures. After testing both C++ and Python prototypes, it showed 10–16× performance improvements in realistic benchmarks and opened the door to structured file modeling, quantum job queues, and spatial scheduling logic.

References

- Python `heapq` docs — `https://docs.python.org/3/library/heapq.html`
- Qiskit SDK — `https://qiskit.org`

- B-tree theory — https://en.wikipedia.org/wiki/B-tree
- GitHub project — https://github.com/AaronCattell/Breaking-the-log-n-Barrier