Sparse based ECS - A Memory-Conscious and Scalable Approach to Runtime Composition

1. Introduction

Entity Component System (ECS) is a data-oriented software architecture that separates data (components) from behavior (systems) and organizes them around lightweight identifiers (entities)—and offers a high-performance alternative to traditional object-oriented design, especially for large-scale simulations and real-time applications. Sparse ECS extends this model with memory-efficient sparse set data structures—a sparse array-backed component layout—and grouping mechanisms that favor data locality and runtime flexibility. This allows for rapid iteration, cache-coherent access, and dynamic reconfiguration of behaviors at runtime.

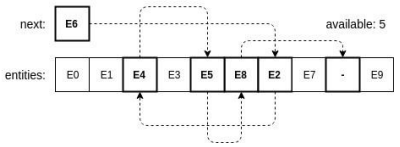
This research presents the design rationale, internal mechanics, and performance benefits of a Sparse ECS, enhanced with insights from an experimental implementation. This document incorporates principles and mechanics drawn from a concrete implementation. Though experimental, the implementation demonstrates key optimizations like swap-based entity recycling, free list management, and linear memory traversal without full allocator integration

2. Entities and Identity Management

Entities in Sparse ECS are 64-bit identifiers composed of two primary fields:

- An index identifying a position in internal data structures
- A version number that ensures safe reuse

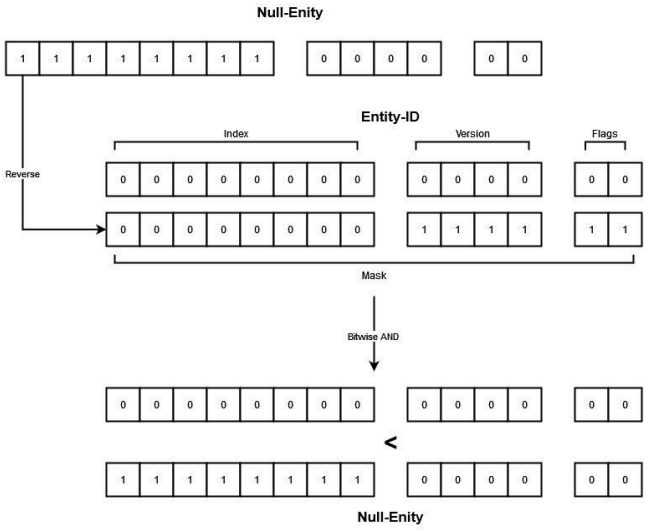
The purpose of the version is to distinguish new entities from recycled ones. Entity reuse follows a deterministic pattern where deleted entities are added to a free list and reissued in FIFO or LIFO order.



Pseudocode Example – Entity Representation:

```
struct Entity {
  uint32 index;
  uint32 version;
}
```

Accessing an entity's data is only permitted if the version matches. Systems querying an outdated entity will detect a mismatch and discard the operation.



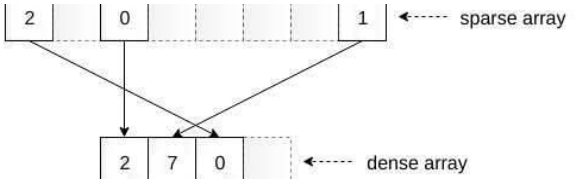
Entity lifecycles are coordinated by a central authority or shard, which tracks allocation, versioning, and disposal. Zones (i.e., separate logical domains for entity management) are not implemented in this version but remain an extension point

3. Sparse Sets and Component Storage

Each component type is associated with a sparse set, a data structure that maps entities to tightly packed arrays.

- The sparse array maps entity indices to dense indices.
- The dense array contains the actual component data.
- A free list tracks available slots.

This approach allows O(1) addition, removal, and lookup, while supporting fast linear iteration over active components.



Pseudocode Example – Sparse Set Access:

```
function get_component(entity):
        if sparse[entity.index].version == entity.version:
                dense_index = sparse[entity.index].dense_index
    return dense[dense_index]
  else:
    return null
```

Internally, when a component is removed, its slot is replaced with the last active element to maintain memory density. The removed slot is then returned to the free list

4. Systems and Stateless Processing

Systems are logic containers that operate over entities possessing specific components. They are stateless, receiving only component pointers as input. The ECS framework determines which entities match the system's requirements and iterates over their data.

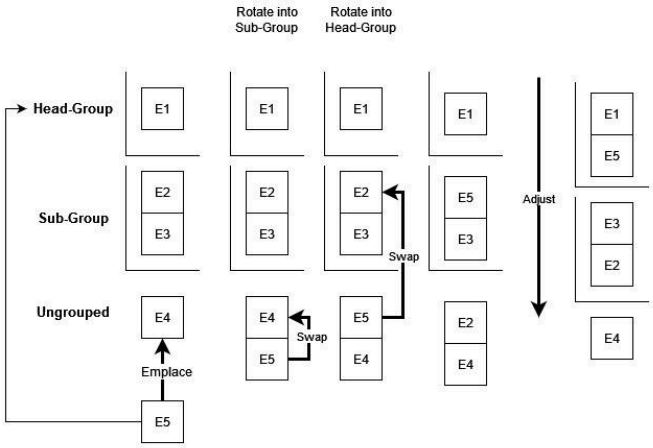
This allows for batch execution, parallelism, and SIMD-friendly memory traversal. Systems are decoupled from allocation and storage, enabling cleaner testing and clearer dependencies

5. Grouping and Memory Layout

Groups organize entities with identical component sets into contiguous memory blocks. Each group maintains a separate

dense array for each component type and a sorted index of matching entities.

When entities change their composition, they may enter or exit groups. This triggers a reordering of components to preserve contiguity and cache performance.



Pseudocode Example – Group Transition:

```
function move_to_group(entity, from_group, to_group):
  for each component in shared_layout:

swap_components(from_group[component], to_group[component], entity.index)
  update_group_indices(entity)
```

Groups are designed to minimize cache misses by ensuring linear traversal over memory. Entities within a group are sorted such that matching components are adjacent in memory.

Subgroups (or slices) allow systems to operate on a subset of components while still benefiting from the main group's layout. They are valid only if their component subset is uniquely identifiable within the main group's layout

6. Entity Recycling and Safety

Entities marked for deletion are added to a free list and their version is incremented. When a new entity is requested, the free list is checked first before allocating a new index.

This ensures:

- Stable memory usage over time
- Protection against stale references via version mismatch
- Minimal allocator pressure

Memory is not reclaimed or freed in the experimental implementation, though such extensions are possible with allocator integration

7. Performance Characteristics

Sparse ECS is optimized for iteration speed and safe dynamic composition:

- O(1) component access through dual indexing
- Linear iteration over densely packed arrays
- Minimal branching, especially in groups
- Version safety during concurrent or deferred removal

These traits make it suitable for real-time systems and large-scale simulations