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.

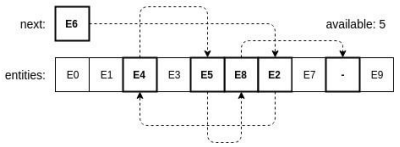
This research presents the design rationale, internal mechanics, and performance benefits of a Sparse ECS, enhanced with insights from an experimental implementation. This document incorporates principles and mechanics drawn from a concrete implementation. Though experimental, the implementation demonstrates 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

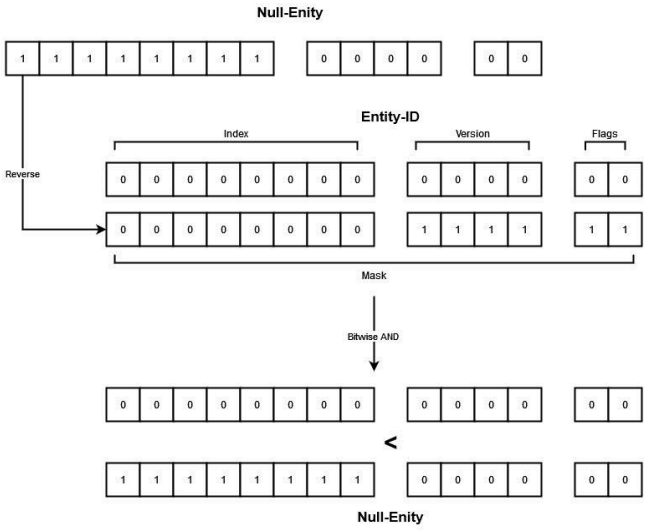
The purpose of the version is to distinguish new entities from recycled ones. Entity reuse follows a deterministic pattern where deleted entities are added to a free list and reissued in FIFO or LIFO order.



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 Systems querying an outdated entity will detect a mismatch and discard the operation.



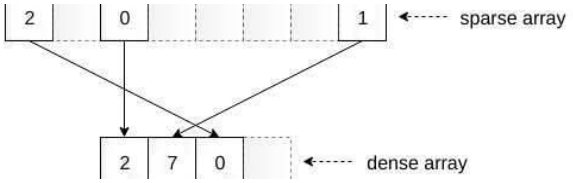
Entity lifecycles are coordinated by a central authority or shard, which tracks allocation, versioning, and disposal. Zones (i.e., separate logical domains for entity management) are not implemented in this version but remain an extension point

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.

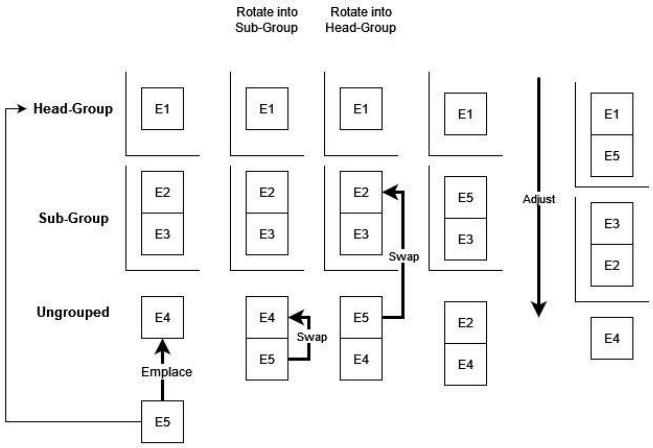
This allows for batch execution, parallelism, and SIMD-friendly memory traversal. Systems are decoupled from allocation and storage, enabling cleaner testing and clearer dependencies

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. Contracts

Entity–Component Systems are optimized for iterating over large collections of entities with similar component compositions. Systems typically operate on groups of components and process them sequentially. While this model is highly efficient for data-oriented processing, it can become inconvenient when a specific entity must be accessed repeatedly or when higher-level abstractions require stable references to a known set of components.

Contracts address this problem by providing a lightweight abstraction that binds a specific component composition to a single entity and exposes direct access to its components. A contract does not replace ECS iteration or grouping mechanisms. Instead, it complements them by offering a stable access layer for cases where direct entity interaction is required

6.1 Contract Forms

Contracts exist in two forms: schema contracts and instance contracts.

A **schema contract** defines a required component composition but does not bind itself to a specific entity. It can be used to validate an existing entity or to construct a new entity with the required set of components. In this form, the contract acts as a structural description of a valid entity composition.

A **contract instance** binds a specific entity to such a schema. It stores the entity identifier and caches access information for the required components. Once bound, the contract provides direct access to those components without requiring repeated lookups in the ECS container.

Schema contracts and instances are independent constructs. A schema may be reused to validate or construct many entities, while instances represent concrete bindings to individual entities

6.2 Non-Exclusive Binding

Contracts are not exclusive. Multiple contracts may reference the same entity simultaneously. The ECS itself does not track or manage existing contracts, nor does it maintain a registry of active bindings.

A contract therefore represents a local abstraction used by higher-level code. It does not participate in the structural management of the ECS. Entities may continue to evolve independently of any contracts that reference them

6.3 Component Access

Instance contracts store access information for each required component. Instead of storing raw pointers, contracts cache the indices of the components within their packed component arrays.

Using indices instead of pointers ensures that cached references remain valid across container reallocations. When a component container grows or shrinks, the base pointer of the packed array may change, but the relative index of an element remains stable. Component references are represented through a generic structure that encodes the required information in a compact form. Similar to entity identifiers, a component reference is stored in a 64-bit value containing the following fields:

---

struct ComponentIndex<T> {
uint32 index;
uint32 version;
}

Internally this structure is represented through a union with a uint64. The structure allows component references to be validated and reconstructed efficiently during access

6.4 Validation and Versioning

Component containers maintain a version counter that increases whenever structural modifications occur. Such modifications include insertions, removals, or internal swaps caused by removal operations.

Contracts use this version information to determine whether cached indices may have become invalid. When the container version changes, the contract refreshes the cached index by performing a lookup for the associated entity.

In addition to the container version, containers may optionally set a dirty flag for the affected entity when structural changes directly impact it. This allows contracts to perform more fine-grained validation when required.

Through this mechanism, the majority of accesses can proceed without performing additional lookups, while still maintaining correctness in the presence of structural changes

6.5 Interaction with Container Operations

Packed component arrays rely on swap-remove semantics. When a

component is removed, the last element of the packed array is moved into the freed position. As a result, the index of that moved component changes.

Whenever such operations occur, the container increases its version counter. If the moved component belongs to an entity that is currently locked or otherwise tracked, the container may additionally set the dirty flag associated with that entity.

Contracts observing a version change can then refresh their cached indices accordingly.

Over time, entities that remain stable tend to experience fewer swaps. As a result, contracts referencing long-lived entities often stabilize and require very few refresh operations. In such cases, component access approaches the cost of direct pointer dereferencing

6.6 Entity Locks

Contracts may rely on entity-level locks to prevent destructive changes to the components they depend on. When an entity is locked, attempts to remove components required by a contract will fail. The container rejects such operations and returns a failure result.

Locks are implemented through flags stored within the entity identifier. Entities may also be locked or unlocked manually by the application code, independently of any contracts

6.7 Structural Independence

Contracts do not guarantee that an entity will continue to satisfy the composition defined by the contract. Entities may gain or lose components over time unless protected by explicit locks.

The ECS itself does not enforce contract validity. Validation is performed only when the contract attempts to access the underlying components.

This behavior differs fundamentally from ECS grouping mechanisms. Groups enforce structural invariants within the ECS itself, while contracts operate purely as external access abstractions

6.8 Foundation for Higher-Level Abstractions

Contracts can serve as foundational building blocks for higher-level abstractions built on top of the ECS.

Examples include:

- Specialized gameplay objects that expose additional methods
- Script objects that contain a contract and a table of callable functions
- Designer or tool objects that provide structured access to specific component sets

In such cases, the contract forms the structural foundation for the object. Additional behavior and logic can be layered on top without changing the underlying data model.

By separating structural access from higher-level behavior, contracts allow complex systems to be constructed without compromising the data-oriented design of the ECS

7. Adaptive Memory Management

One of the fundamental design principles of the presented architecture is the complete separation between logical composition, data access, and physical storage. Neither Contracts nor Groups prescribe how component data must be organized in memory. Instead, physical storage is delegated entirely to the memory manager. This separation enables an adaptive storage layer capable of optimizing memory locality independently of the ECS itself.

Unlike archetype-based ECS implementations, where entity composition directly determines storage layout, the proposed architecture allows memory organization to evolve independently from logical composition. Consequently, storage optimization becomes an implementation detail rather than a structural property of the ECS

7.1 Chunk-Based Storage

Rather than storing every component into continuously growing dense arrays, groups may be partitioned into fixed-size chunks if applicable. Each chunk stores tightly packed Structure-of-Arrays (SoA) data for the components owned by the corresponding group.

Systems iterate sequentially over the logical chunk sequence. Since chunks represent independent storage units, the memory manager may relocate them transparently without affecting contracts, groups, or system implementations. Choosing a fixed chunk size of for example 64KB, aligns naturally with modern cache hierarchies and page sizes while avoiding the disadvantages of continuously growing allocations

7.2 Profile-Guided Memory Placement

The physical placement of chunks may be optimized using memory profiles. Profiles describe expected access locality and may either be defined explicitly by the developer or generated automatically through runtime profiling.

A developer-defined profile may specify the expected execution frequency, component locality or frequently shared components. Static profiles provide an initial layout while runtime statistics allow the memory manager to gradually refine placement according to actual workload characteristics if needed.

This concept is comparable to compacting garbage collectors in managed runtimes. Rather than relocating objects, however, the memory manager relocates complete component chunks to improve spatial locality while preserving logical ECS structures

7.3 Materialized Partial Views

Some components naturally participate in many independent systems. A typical example may be the Transform component, which may be required simultaneously by rendering, animation, physics, audio and gameplay systems.

A straightforward solution would be the creation of increasingly larger groups containing every commonly used component. Such an approach, however, gradually reintroduces the structural coupling commonly found in archetype-based ECS implementations.

Instead, each group remains responsible only for the components it owns.

To improve locality for frequently shared components, the memory manager may reserve a virtual cache region inside every chunk. Before a system begins iterating, the required shared components may be materialized into the reserved cache region if needed. The resulting layout allows the system to iterate over a contiguous memory region containing both the owned components and the cached view while leaving the original component ownership unchanged.

Since these views are temporary, they may be discarded, regenerated or reused whenever necessary. This approach combines the flexibility of sparse component ownership with locality approaching that of specialized archetype layouts without introducing permanent data duplication or tighter structural coupling

7.4 Adaptive Layout Optimization

Because chunk placement is entirely controlled by the memory manager, physical storage may continuously adapt during runtime. Frequently accessed chunk sequences can be compacted into contiguous memory regions, while rarely accessed data may remain fragmented without affecting execution performance.

Similarly, chunks belonging to related groups may be placed close together when profiling indicates repeated sequential access. Unlike traditional ECS implementations, these optimizations require no changes to the underlying architecture but become an adaptive optimization layer capable of responding to changing runtime behavior instead

8. 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

9. 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
- Management strategies to allow optimized cache locality

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