An Investigation of Data Storage in Entity-Component Systems

Bailey V. Compton

AN INVESTIGATION OF DATA STORAGE IN ENTITY-COMPONENT SYSTEMS

THESIS

Bailey V Compton, 2nd Lieutenant, USAF

AFIT-ENG-MS-22-M-018

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-22-M-018

AN INVESTIGATION OF DATA STORAGE IN ENTITY-COMPONENT SYSTEMS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Science

Bailey V Compton, B.S.C.S.

2nd Lieutenant, USAF

March 24, 2022

AFIT-ENG-MS-22-M-018

AN INVESTIGATION OF DATA STORAGE IN ENTITY-COMPONENT SYSTEMS

THESIS

Bailey V Compton, B.S.C.S.
2nd Lieutenant, USAF

Committee Membership:

Douglas D Hodson, Ph.D
Chair

Major Richard Dill, USAF, Ph.D
Member

Michael R Grimaila, Ph.D
Member

AFIT-ENG-MS-22-M-018

Abstract

Entity-Component Systems (ECS) have grown vastly in application since their introduction more than 20 years ago [1]. Providing the ability to efficiently manage data and optimize program execution, ECSs, as well as the wider field of data-oriented design, have attained popularity in the realms of modeling, simulation, and gaming. This manuscript aims to elucidate and document the storage frameworks commonly found in ECSs, as well as suggesting conceptual connections between ECSs and relational databases. This formal documentation of the in-memory storage formats of entity-component systems affords the United States Air Force, the Department of Defense, and the software engineering community a greater understanding and applicability of the paradigm.

AFIT-ENG-MS-22-M-018

For my parents.

Thanks for always answering the phone and never asking how school is going.

Acknowledgments

I would like to thank my advisor, Dr. Doug Hodson, for his constant support and guidance throughout this research. Your knowledge and initiative have been invaluable to me.

I would also like to thank my other committee members, Dr. Michael Grimaila and Maj Richard Dill, for their input and assistance with this document.

Table of Contents

AN INVESTIGATION OF DATA STORAGE IN ENTITY-COMPONENT SYSTEMS

I. Introduction

1.1 Problem Background

In 2009, Noel Llopis published an article titled "Data-Oriented Design (Or Why You Might Be Shooting Yourself in the Foot with OOP)" [2]. In this article, Llopis expounds upon the need for software development to shift from a mindset of object-oriented programming (or procedural programming, functional programming, or any other paradigm) to a data-first mindset. In his proposed paradigm, data is the central foundation of any program. This is not a revolutionary idea - storage, referencing, and manipulation of data is the impetus for creating any program. Data-oriented design of programs, however, focuses solely on how the execution of the program changes the data. Then, functionality is created to match the desired changes.

Since the publishing of the 2009 article, data-oriented design has grown in popularity, gaining traction in game engines and other applications that require vast amounts of data. As popularity has grown, documentation has as well. In 2018, Richard Fabian published a book on data-oriented design, documenting the need for data-oriented design and detailing best practices when creating an application using this paradigm [3]. Early in the text, Fabian includes a chapter regarding relational databases, discussing the fact that "techniques we used to use to defend against latency from CPU to hard drive, now apply to memory" [3]. It was this chapter in a data-oriented design book that motivated the research into the relationship between

Entity-Component Systems (ECS) and databases conducted in this thesis.

When initially reviewing resources to begin this research, it was found that there were few academic and/or peer reviewed sources relating to data-oriented design, and even fewer relating specifically to entity-component systems. Recognizing the need for documentation in this field, this thesis intends to fill a gap in documentation in a quickly growing field in software engineering.

1.1.1 A Brief Introduction to Entity-Component Systems

ECS are a paradigm of data-oriented design that specifically separate program behavior from program data. Entities and components are abstractions of the program data. The system represents program functionality itself.

Within an ECS, components are the actual values of program data that will be manipulated. Components can be represented by any data type or data structure, but work most efficiently when in the most atomic form possible. Components are the data that motivates program execution.

Entities are the abstract concepts by which components are grouped and referenced. Entities are normally simple identifiers.

The system within an ECS refers to the behavior and functionality within the program that uses and manipulates the data stored in components and referenced by entities.

Entity-component systems further separate program behavior from program data by requiring data to be accessed through a query system.

1.1.2 ECS Applications

ECSs can be utilized in essentially any program or application. However, the computational efficiency, especially in data-intensive applications, afforded through

ECS-based systems makes it popular in the fields of gaming and simulation.

The 2018 National Defense Strategy (NDS) outlines the strategic approach for development of Department of Defense (DoD). One of the three lines of effort for national defense requires the United States to "Build a More Lethal Force" [4]. In direct support of this line of effort, the DoD needs to modernize key capabilities, including Command, Control, Communications, Computers and Intelligence, Surveillance, and Reconnaissance (C4ISR) and the warfighting domains in space and cyberspace. ECSs can be leveraged to support these efforts through effectively modernizing aging DoD systems and optimizing execution of current functional needs. The DoD also employs data-intensive applications, such as simulated aircraft trainers, that would benefit from any optimization possible.

1.1.3 ECS Libraries

Although there were only three Entity-Component System (ECS) libraries selected for survey in the second paper in this thesis, eight were considered. These implementations are outlined in this section, as well as why they were or were not selected for further research.

1.1.3.1 Bevy

Bevy, written in Rust, provides a fully-equipped game engine using a custom ECS system. Library functionality utilizes many structures and functions native to the Rust language, making it both efficient and easy to understand. The underlying ECS system providing data storage to Bevy was previously based of the Hecs library, but has since been split from Hecs to its own ECS implementation [5].

Bevy was selected for examination in the survey because of its robust functionality and its multiple storage frameworks with which to organize data.

1.1.3.2 EnTT

EnTT is a concise, header-only C++ ECS library. Notably used in the popular video game Minecraft, EnTT provides robust functionality on top of entity-component storage. The complexity of implementation of EnTT resulted in its removal from the list of candidates for this research. [6]

1.1.3.3 Esper

Esper is a compact ECS library written in Python. It drew interest for this research given its small codebase and novel implementation. While it does not have quite the extensive documentation boasted by the other libraries selected for survey, it does provide interesting insight into the possibilities of implementations of ECS outside of the standard sparse set and archetype. [7]

1.1.3.4 Flecs

Flecs was created in C as an efficient entity-component system for games and simulations with large numbers of entities. It features a native Application-Programmer Interface (API) for integration with "most game engines and scripting languages." Flecs specifically markets its use of archetype storage and structure of arrays organization to process millions of entities at each iteration of execution [8].

Flecs was ultimately chosen to be surveyed due to its in-depth documentation, as well as its involved development community and extensibility.

1.1.3.5 Legion

Legion is a popular ECS implementation in Rust, offering many of the same functions as Bevy or Shipyard. It provides out-of-the-box functionality to developers, with "minimal boilerplate." However, Bevy had already been selected for this research, and it was determined that the two were similar enough implementations that only one was necessary to be surveyed. [9]

1.1.3.6 Hecs

Hecs (a handy ECS) was considered as a possible Rust implementation for this research. The hecs library offers an uncluttered ECS implementation. Hecs provides little guidance on the organization of data stored within the system, instead opting for entirely developer-defined organization.. Due to the nature of the survey conducted in this research, it was ultimately decided that other implementations would better suit the requirements. [10]

1.1.3.7 Shipyard

Shipyard was created as a competitor to Specs in the Rust-based ECS space. Shipyard follows a data organization structure similar to EnTT. It was eventually removed from the list of candidates for this research because its implementation is derived from other ECS libraries already considered as candidates. [11, 12]

1.1.3.8 Specs

Specs, a Rust-based ECS library, was also considered for this research. It provides great extensibility and the ability to add functionality to the ECS itself while still utilizing the library. Specs supports five different storage frameworks, none of which exactly line up with the more common archetype or sparse set organizations. It was decided that the addition of more storage frameworks would be too broad for the scope of this particular endeavor. [13]

1.1.4 The Storage Dilemma

Because the program data is the foundation of entity-component systems, the storage of that data requires thoughtful planning. Array storage maximizes the potential for Single Instruction, Multiple Data (SIMD) processing of data, which, in turn, maximizes potential operating efficiency [14]. There are two possible ways to implement array storage - Array of Structures (AoS) or Structure of Arrays (SoA) [14]. Both AoS and SoA have advantages and disadvantages. There is not one that invariably outperforms the other.

Structs are developer-defined data structures that combine multiple existing data types to be treated as a single structure within the program [15]. Data structures contained within a struct are neighboring in memory - that is, data within the struct is contiguous. Both AoS and SoA leverage the ability of structs to store varied data types neatly and contiguously in memory.

Storage organized based on a SoA framework creates a single struct to hold multiple arrays of data. The arrays within the struct are not required to be the same length or hold the same data type. In this organization, all data of a certain type or use in the struct is stored contiguously in memory. An example implementation of a simple structure of arrays is shown in listing I.1.

Listing I.1: Psuedocode implementation of Structure of Arrays

```
struct SoA {
    A a[100];
    B b[100];
    C c[100];
};
```

A structure of arrays organization can be beneficial when a program only requires certain components from within an entity. When the query for those components occurs, only the necessary arrays of components are loaded into the CPU cache, saving clock cycles and cache space when compared with loading all components.

The AoS frame work organizes data by grouping each set of related data into a struct and creating an array of that struct. As such, the data within each struct is contiguous, meaning that data of the same type are stored every *nth* type in memory, where n is the number of data types in the structure. In other words, all data types within a certain struct are stored together, before the next struct is stored as a neighbor. As example implementation is shown in listing I.2

Listing I.2: Psuedocode implementation of Array of Structures

```
struct AoS {
    A a;
    B b;
    C c;
};
AoS aos[100];
```

Figure 1 visualizes the in-memory arrangement of a block of data in both AoS and SoA formats. The array of struct[ure]s in the figure stores each datum sequentially, by structure, such that the array stores data in a repeating pattern of data types. The struct[ure] of arrays clearly shows that all data of a certain type is stored contiguously before the next data type is stored.

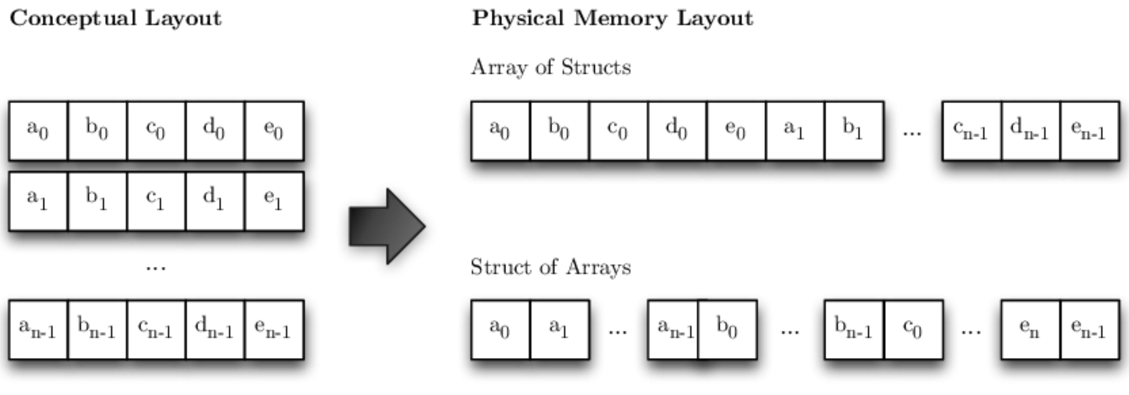

Figure 1: Diagram of AoS and SoA organizations in memory [16]

1.2 Research Objectives

This thesis aims to elucidate the mechanisms by which data is organized and stored within entity-component systems. To achieve this goal, the following objectives were outlined:

- Document the common storage models used by entity-component systems.
- Recognize the similarities between relational databases and ECSs.
- Discuss the in-memory organization of ECS frameworks.

1.3 Document Overview

This thesis investigates different aspects of entity-component systems. This introduction gives an overview of the problem domain that prompted this research, and the goals that guided the course of the research.

Following chapter I, there are two article-style papers discussing different topics of entity-component systems. The first, a conference-length article, discusses the relationship that can be found between entity-component systems and relational databases. This research can be found in chapter II.

The second paper, in chapter III, surveys three open-source entity-component system libraries to better understand how data is stored within programs that employ ECS storage.

Chapter IV synthesizes the previously discussed research in chapters II and III to document the nature of entity-component systems and consider future research on the topic.

II. Paper I: Comparison of Archetypal Entity-Component Systems and Relational Databases

The following paper, “Comparison of Archetypal Entity-Component Systems and Relational Databases,” was submitted and accepted by the 2021 World Congress in Computer Science, Computer Engineering, and Applied Computing; it was published in July of 2021.

Comparison of Archetypal Entity-Component Systems and Relational Databases

Bailey V. Compton, Douglas D. Hodson, Richard Dill, and Michael R. Grimaila
Air Force Institute of Technology, WPAFB, OH, USA
emails: bailey.compton@afit.edu, doug@sidechannel.net, richard.dill@afit.edu, michael.grimaila@afit.edu

Abstract— *The Entity-Component-System (ECS) architectural design pattern separates data from computer logic (i.e., behavior) - components define data, systems define behavior. As such, it embraces the ideas espoused by Data-Oriented Programming. Relational databases can be compared to ECS implementations through emphasis on organization and retrieval of data. Conceptualizing archetypal ECS tables as single-table, standalone databases allows for Data-Oriented programmers to employ database standardization practices to the organization of data in ECS applications.*

Keywords: ECS, Entity-Component System, Relational Databases, Data-Oriented Design, Data-Oriented Programming

1. Introduction

The mapping of objects for application development has required more and more overhead as the requirements for applications change over time [1]. In order to combat the growing workload required by object-oriented programming, a new paradigm has emerge - data-oriented design. As data-oriented development grows in usage and popularity, efforts must be made to connect data-oriented design principles to existing research to expedite the investigation of the applications of data-oriented applications.

2. Background

2.1 Data-Oriented Design

All programs exist to transform data [2]. Many frameworks and paradigms conceptualize how data should be manipulated and transformed within an application. Object-oriented programming (OOP) gained popularity as a development paradigm, as it allows developers to envision data as objects that interact with other objects by exchanging messages. OOP combines the data that is being transformed with a set of functions that manage it. In contrast, data-oriented design (DOD), also referred to as data-oriented programming (DOP), emphasizes separation of data and the transformation thereof. Under DOD, data encapsulation is minimized as much as possible, instead opting for plain old data (POD) to be stored as simply and efficiently as possible. OOP couples the problem domain directly with the data, which can lead to issues with extensibility when applications need to be updated. Data-oriented design lacks such flaws, since data remains separate from implementation [3]. DOD also tends to be more efficient when operating on large amounts of data. Data-oriented design has been shown to outperform object-oriented programming in modeling & simulation applications, which execute on copious amounts of data [4].

2.2 Entity-Component Systems

Entity-component system (ECS) is a software design pattern that can be classified as a form of data-oriented programming. ECS separate data from system execution, organizing information elements into entities, which consist of a set of components. Data that is to be read, stored, and manipulated by a system uses queries. Entity-component system approaches are favored by game and simulation developers, especially, because entity-component systems tend to handle large amounts of game objects (data) well, and offer easy extensibility [5].

2.2.1 Data Querying

Entity-component systems use queries to interface with organized data, in order to create, read, update, or delete the data it is referencing within the application – but queries do not provide any extra functionality. Any functionality required past data storage and retrieval is the responsibility of the application itself. Querying the data to return components or entire entities allows the developer to organize data in the most efficient fashion for their specific application. Advance knowledge of how data will be referenced and manipulated creates greater opportunity for optimal execution of the system as a whole.

2.2.2 Data Storage

There are several forms of implementations of entity-component systems. Forms differ in the organization of data within program memory, while access mechanisms to the data remain the same. The two most popular storage organizations are sparse set and archetypal.

Sparse set ECS is optimized for applications that may have many entities, but there are few or no entities that contain the same components. In a sparse set implementation, components are stored in arrays of each data type, with simple unsigned integer indices referencing the entity the component corresponds to. This structure benefits applications that require a lot of insertion or deletion of components, since those operations can occur in near constant time. The trade-off, however, is that the random access patterns required for reading or manipulating these data are inefficient [6], [5].

Archetypal ECS, in contrast, benefits applications with many similar entries, with the same components. In an archetypal implementation, entities are stored as rows in a table, which is called an archetype. Each entity in an archetype contains the same components, with each component stored as a column in the table. There may be one or many archetypes in the system, depending on the specific requirements of the application. The arrangement of data according to entity allows for query evaluation and entity retrieval operations to occur in constant time, on average [5], [6]. However, the search and data movement required for insertion and deletion of entities or components makes those operations more expensive than the sparse set implementation. For this research (i.e., comparison to relational databases), the ECS storage mechanism of interest is archetypal, since it seems to closely resemble the organization of a database table.

2.3 Relational Databases

Relational databases have long been popular for storing and using large amounts of data, from online storefronts to medical records. In a relational database, information is arranged in tables, where each row represents a single record and each column represents a data field that record should contain. Each table in the database will have one or more fields that also occur in another table. These fields are keys, which allow users to connect records through queries to the database. Databases logically organize data in a form that allows users to retrieve the exact information they require with little knowledge of the content or organization of the database itself.

2.3.1 SQL

In order to access information in a relational database, users must execute a query. Over the years that relational databases have developed, many query languages have evolved, but most query languages in use today are based on the Structured Query Language (SQL). SQL queries can create, read, update, and delete information from tables, and can also combine data from multiple tables, using key fields to combine records. SQL also offers many time-saving functions to build into queries, such as number rounding, text formatting, and finding sums and averages of columns [7], [8].

2.3.2 Database Normalization

Database normalization serves to limit the probability of data inconsistencies and redundancies. Normalizing a database aims for each item of data to be stored in the database once, because any more stores of the same information would be superfluous. The common levels of normalization are briefly outlined below. Each level builds on the previous, so any database normalized to a certain level is also normalized to every level below it.

First normal form (1NF) is fulfilled by any relational database. The only requirement of 1NF is that all records in a table contain the same number of fields, and each field may only contain a single atomic datum. What constitutes an atomic value for each field may be left to the structure and use of the database [9].

Second normal form (2NF) requires records using composite keys (keys consisting of more than one field) to not contain any fields that relate to a subfield of a key. To fulfill 2NF, tables that contain information specific to a subfield of a key should be divided into multiple tables, separating information about the key's subfield into its own table. Tables that do not employ composite keys and fulfill 1NF automatically fulfill 2NF [9].

Third normal form (3NF) requires all nonkey fields in a record to not depend on any other nonkey fields. If there are nonkey fields in a record that can be logically based on other fields, the record should be decomposed into two records, storing related information in a separate table that can be referenced by a single reference fact in the original table. [9] There is a more restrictive form of 3NF referred to as Boyce-Codd normal form (BCNF), that further limits redundancies, but has potential to not protect dependencies [10].

Fourth normal form (4NF) requires fields within a table to be logically related to all other fields within the table. If two facts within a record are not related, they should be decomposed into two separate tables [9].

Fifth normal form (5NF) attempts to generally encompass decomposition of information that is not addressed in second, third, or fourth normal forms. 5NF is fulfilled if and only if every piece of information possibly needed to be queried can be created by join functions between tables [11].

Normalization does require a trade-off, however. In order to reduce redundancy, data that is usually queried at the same time (such as a field for street address and a field for city) may be split to create the most "atomic" field entry possible. This means that in order to retrieve an address record, two fields must be pulled instead of one – every time. However, depending on the uses of the database, this data can be denormalized and combined to optimize performance. Permanent linkages in data constitute atomic values, but determination of what linkages are permanent is subjective to the application of the database in question.

3. Relationship between ECS and Relational Database Tables

It is simple to see the comparison between an archetypal implementation of ECS and a relational database. The basic functions of a data retrieval system are creation, reading, updating, and deletion. Both entity-component systems and relational databases, as data retrieval systems, provide these basic functions. The organization of ECS entities into tables, with table makeup determined by the components defined by an entity, parallels the organization of database tables in first normal form. In fact, archetypes can be conceptualized as single-table databases. Each table within an archetypal ECS can be thought of as a standalone database, that is queried and may benefit from normalization. There is potential, depending on the application, for the size of the database to grow past a single table, but that depends on the implementation of the ECS framework on a case-by-case basis.

3.1 Data Queries

One of the significant factors that facilitates a comparison between an archetypal ECS and a relational database is the querying of data from the system. While SQL and its ilk may provide additional functionality for ease-of-use, both database query languages and ECS queries provide the same four main operations – create, read, update, and delete. These are the four operations of any information storage and retrieval system, and anything else built into a query is extra. Both forms of queries also allow for retrieval of multiple entries at once, up to an entire table, as opposed to a single record at a time, iteratively. One difference in the query functionalities of ECS and databases is the join and union functions. SQL allows for the resulting table of data (that is a combined view of several joined disjoint tables), using queries. It requires minimal knowledge of the tables (only the necessary fields) to create a new record or field from two or more tables. However, in the most generalized form of this comparison, each archetype in the ECS is viewed as its own, standalone database. As such, the need for union or join operations is nonexistent.

3.2 Normalization

One of the most important functions of a relational database is limiting data redundancy with normalization. If ECS is viewed as a database, it logically follows that the tables within the ECS should be normalized. If an ECS framework is used for a modeling and simulation application, such as a flight model, there is no reason to duplicate information about the aircraft being modeled. Querying extra information will just waste clock cycles and storing that information will waste storage space. Database normalization focuses on normalization of records of a single table, so a normal form may be applied to an ECS table with minimal adjustment of thought. Normalization provides a standard by which archetypes can be developed and judged. Database normalization has been practiced for more than thirty years, and many references and examples are readily available to ECS developers that can be used for organizing and normalizing their archetypes.

4. Conclusions

The concept of relational databases was first published in 1970 [12], and the first paper proposing a standardized query language followed quickly after in 1974 [7]. In the more than fifty years since the conceptualization of relational databases, usage, development, and research in the field has risen exponentially. Connecting the inner workings of entity-component systems and relational databases allows data-oriented design users and developers to benefit from the depth of knowledge of database systems. This shift in mindset for ECS, starting with the archetypal storage format, opens the door for potential leaps and bounds of growth of knowledge in applications using ECS.

5. Disclaimer

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the U.S. Government.

References

[1] T. Neward, “The Vietnam of Computer Science,” 2006. [Online]. Available: http://blogs.tedneward.com/post/the-vietnam-of-computer-science/

[2] M. Acton, “Data-Oriented Design and C++,” in *Cpp-Conference*, 2014. [Online]. Available: https://www.youtube.com/watch?v=rX0ItVEVjHc

[3] R. Fabian, *Data-Oriented Design*, C. Ring, Ed., 2018. [Online]. Available: www.dataorienteddesign.com

[4] J. Vagedes, “A study of execution performance for rust-based object vs data-oriented architectures,” Ph.D. dissertation, Air Force Institute of Technology, 2020.

[5] S. Mertens, “ECS FAQ,” 2021. [Online]. Available: https://github.com/SanderMertens/ecs-faq

[6] C. Anderson, “Bevy 0.5,” p. 2021, 2021. [Online]. Available: https://bevyengine.org/news/bevy-0-5/

[7] D. D. Chamberlin and R. F. Boyce, “Sequel: A Structured Query Language,” IBM Research Laboratory, San Jose, California, Tech. Rep., 1974.

[8] D. D. Chamberlin, “Early history of SQL,” *IEEE Annals of the History of Computing*, vol. 34, no. 4, pp. 78–82, 2012.

[9] W. Kent, “A simple guide to five normal forms in relational database theory,” *Communications of the ACM*, vol. 26, no. 2, pp. 120–125, 1983.

[10] R. Wakefield, “BCNF and 3NF,” Colorado State University, Fort Collins, CO, Tech. Rep., 2020.

[11] R. Fagin, “Normal forms and relational database operators,” IBM Research Laboratory, San Jose, California, Tech. Rep., 1979.

[12] E. F. Codd, “A Relational Model of Data for Large Shared Data Banks,” *Communications of the ACM*, vol. 26, no. 1, pp. 64–69, 1983.

Author Biographies

BAILEY V. COMPTON is a 2nd Lieutenant currently stationed at Wright Patterson Air Force Base. She is Master’s Student at the Air Force Institute of Technology (AFIT) studying Computer Science with an emphasis in Software Engineering. Lt Compton is a 2020 graduate of the United

States Air Force Academy, where she studied Computer Science.

DOUGLAS D. HODSON is an Associate Professor of Computer Engineering at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Physics from Wright State University in 1985, and both an M.S. in Electro-Optics in 1987 and an M.B.A. in 1999 from the University of Dayton. He completed his Ph.D. at the AFIT in 2009. His research interests include computer engineering, software engineering, real-time distributed simulation, and quantum communications. He is also a DAGSI scholar and a member of Tau Beta Pi.

MAJOR RICHARD DILL is an Assistant Professor of Computer Science at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Computer Science from the University of Maryland at College Park in 2004, and both an M.S. in Computer Science in 2008 and a Ph.D. in 2018 from AFIT. Major Dill's research interests include computer security, algorithms, and artificial intelligence.

MICHAEL R. GRIMAILA (BSEE 1993; MSEE 1995; Ph.D. 1999, Texas A&M University) is a professor and head of the Department of Systems Engineering and Management at the Air Force Institute of Technology, Wright-Patterson Air Force Base in Ohio, USA. He is a member of Tau Beta Pi, Eta Kappa Nu, and the Association for Computing Machinery, as well as a Senior Member of the IEEE, and a Fellow of the Information System Security Association. He can be contacted via email at michael.grimaila@afit.edu.

III. Paper II: Survey of In-Memory ECS Contiguous Storage Types

The following paper, “Survey of In-Memory ECS Contiguous Storage Types,” will be submitted to an appropriate journal at a later date.

Survey of In-Memory ECS Contiguous Storage Types

Bailey V. Compton, Richard Dill, Douglas D. Hodson, and Michael R. Grimaila
Air Force Institute of Technology, WPAFB, OH, USA
emails: bailey.compton.1@us.af.mil, richard.dill@afit.edu, doug@sidechannel.net,
michael.grimaila@afit.edu

Abstract—*Entity-component systems continue to grow in popularity and relevance in the software development space. However, there is little academic reference available documenting the commonly accepted forms of storage within these systems. This research attempts to mitigate that void of information by surveying three open-source entity-component system libraries, implemented in three different languages, to ascertain the in-memory organization of data in these applications. Two common forms of data organization were investigated - archetype and sparse set - and are also included in this research.*

1. Introduction

As computer applications are required to process more and more data at ever increasing speeds and degrees of efficiency for normal execution, developers and engineers are faced with two options to fulfill the requirements. The first is to increase the hardware efficiency of computing machines, which computer engineers have been able to achieve with some success. The second line of effort is to optimize the organization of data before the program executes, to the benefit of the hardware. This organization of back-end application data requires prior consideration from the developer to construct a schema with which to store and retrieve information.

Prior consideration of data organization is a mindset shift from the pervasive ideas of today. Object-oriented programming, which interweaves data and functionality in the mind of the developer, is widely taught and used in computer science. Data-oriented design of programs (also referred to as data-oriented programming) treats the information manipulated in a program as completely separate from behavior of the program [1]. Any application simply retrieves, manipulates, and stores information, instead of viewing every aspect of the program as an object that can manipulate data and/or be manipulated.

This paper investigates Entity-Component Systems (ECS),a paradigm of data storage and retrieval in memory for applications. This research details this paradigm, analyzes several popular implementations thereof, and compares them to generate a deeper understanding of the pattern as a whole.

2. Entity-Component Systems

The entity-component system is divided into three distinct segments - entities, components, and the system. Data used in an ECS-based program constitutes the components. These components ideally exist in their most atomic form, commonly a plain data type, such as a floating point number or integer. However, if more than one individual datum will be always be used and/or referenced at the same time, they may be combined into a single component consisting of a single complex data structure.

Entities within the program serve as the method of referencing and conceptually grouping components - an entity serves as a unique identifier that can be applied to components. Entities may be connected to an unlimited number of components. These associations are determined by the needs of the application. The functions used by the application constitute the system, which defines behavior in relation to entities based on components. Systems access components for reference and modification through queries [2].

An example configuration of an ECS is visualized in Figure 1. In this example, Entities A and B both reference four components - Translation, Rotation, LocalToWorld, and Renderer. Entity C only references Translation, Rotation, and LocalToWorld. The system references the components Translation and Rotation, conducts a behavior (in this case, simple multiplication), and stores the result into the LocalToWorld components.

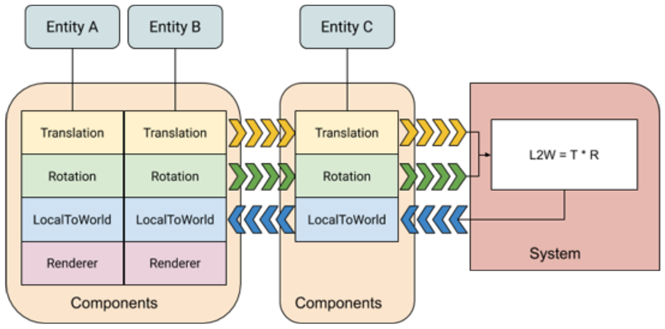

Fig. 1: Block Diagram of Entity-Component System [3]

Many ECS implementations organize components contiguously in memory. The use of contiguous storage allows for the use of Single-Instruction, Multiple-Data (SIMD) parallel processing. The use of SIMD processing allows for many similar data elements to be operated on simultaneously, as opposed to sequentially [2]. The act of enabling and using SIMD processing on a data vector is referred to as vectorization. In data-intensive applications, the impact of simultaneous execution is easily observed in speed and efficiency of execution [4]. Because the developer is determining the configuration of application data within memory, neighboring data is much more likely to be used at similar times. This minimizes cache misses and makes referencing components even faster. There are several forms of internal data storage configurations that are popular in ECS systems. The two most popular in-memory storage frameworks for ECS implementations are archetype and sparse set.

2.1 Archetype

Storing components within an ECS is a complex problem. It is beneficial for program efficiency to store like components together, which maximizes the potential for SIMD processing in contiguous arrays [5]. However, it logically follows that the components of a certain entity will commonly be used together, so it would be more efficient to store components by entity. Organizing components contiguously is not a trivial task.

As an example, take a program with 3 entities, all with 2 components, a and b. Two arrays can be created, one storing all a components and one storing all b components. In this case, array indexes can be used as entity identifiers, since both component arrays are parallel.

Now suppose that component b must be removed from the entity at array index 1. The array holding the b components is no longer contiguous, which means it cannot be vectorized, or use SIMD (see Figure 2).

```
0: [A B]
1: [A  ]
2: [A B]
```

Fig. 2: Diagram of a and b arrays after removal of $b[1]$ [5]

It is a trivial set of operations to make these components contiguous in memory again. The entities at indexes 1 and 2 are swapped, resulting in the following organization (see Figure 3) [5].

```
0: [A B]
2: [A B]
1: [A  ]
```

Fig. 3: Diagram of a and b arrays after swapping storage of indexes 1 and 2 [5]

As long as there are only 2 components that need to be stored, components can be stored in a vectorizable configuration (Figure 4). However, as soon as a third component is added to the system, contiguous storage becomes significantly more complex. If every combination of components is present in the system's entities, there is no way to store all components of a type contiguously (Figure 5).

```
3: [  B]
0: [A B]
2: [A B]
1: [A  ]
```

Fig. 4: Diagram of 2-component contiguous storage [5]

```
0: [  B  ]
1: [  B C]
2: [A B C]
3: [A B  ]
4: [A    ]
5: [A   C]
6: [    C]
```

Fig. 5: Diagram of 3-component storage [5]

This problem can be extrapolated to any set of components larger than 2. Dubbed the "ABC Problem," developers needed a different strategy with which to store components in memory.

As a solution to the "ABC problem" [5] archetype-based storage systems organize data into “tables” [2], [6], [7]. A table can be visualized as a spreadsheet or an individual table in a relational database. Every column of the table contains one component, so that each individual component within the column is of the same data type as all others in the column. Each row of the table corresponds to a specific entity within the system, such that each entity relates to one of each component it intersects in the table.

In order for this structure to work correctly, each table only contains one type of entity – that is, each specific combination of components that make up an entity is given its own unique table. Since each combination of component types only correspond to one table, there is no duplication of entities between tables.

This organization of components is illustrated in Figure 6. In this example, there are four archetypes. The first contains all components for entities A, B, C, and D. These four entities each contain two components - 1 and 2. The subsequent tables follow the same pattern, with $1 - \infty$ entities assigned to the archetype and $0 - \infty$ components assigned to each entity. If there are no entities stored in an archetype, the table does not exist. As shown in Figure 6, archetypes can overlap in what individual components they contain, but each table has a unique combination of components. Every entity will only be stored in one archetype table.

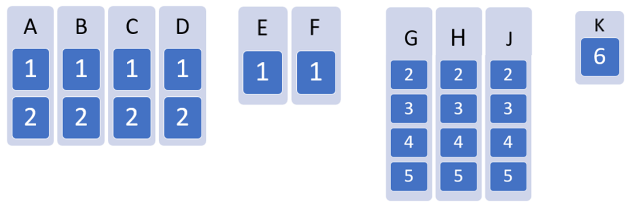

Fig. 6: Block Diagram of Archetype Tables in an ECS

This tabular organization of data allows for swift return of query results and quick iteration of components within program execution [7]. This speed of referencing data comes with drawbacks. Due to each type of entity having its own archetype (table), addition or deletion of components on a specific entity, or addition of an entire entity, is an expensive operation. Any change made to an entity requires the application to search for the correct archetype for the new entity, create a new archetype (if necessary), add the new entity to the correct table, and remove the old entity from the previous table.

To facilitate transfer of entities between archetypes, it is common for entity-component systems to implement an archetype graphing system [5]. Figure 7 shows an example of such a graph. In the example, each edge of the graph represents a component that can be added or removed from the node (archetype) it is leaving, depending on whether the component of that edge is present in the table.

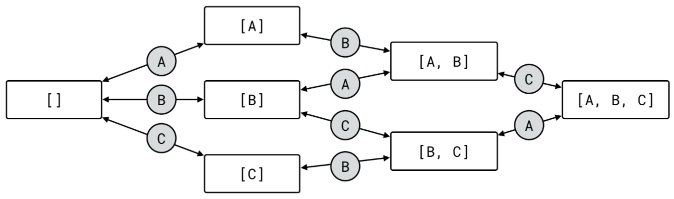

Fig. 7: Diagram of Archetype Graph [5]

2.2 Sparse Set

Archetypal storage systems make for quick retrieval and iteration of data, but some applications value addition and removal of components over retrieval. In that case, a sparse set storage framework would be more beneficial.

While archetypal storage systems organize data by entity, sparse set storage systems group data by component. Every component of the same type is grouped into arrays and stored in program memory. In order to efficiently manage this data, a sparse set data structure is used. Sparse set arrays are not single, simple contiguous vectors. A sparse set data structure is instead two arrays, one densely packed and one sparsely populated [8], [9], [10], [11].

The two vectors within the sparse set structure are based on each other. The densely packed array contains integers, which are entity identifiers. The order of entities within this array is arbitrary. The sparsely packed array contains integers, such that the data stored in $sparse[n]$ is the index of where the entity identified by n is stored in the dense array. In other words, the index of a certain datum in the sparse array is the entity ID and the datum in that index is the location of that entity ID in the dense array [12]. This concept is visualized in Figure 8.

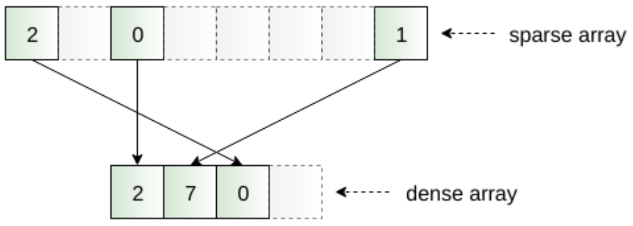

Fig. 8: Diagram of sparse set arrays [8]

By storing entity information in two arrays containing the same volume of data, the program trades memory efficiency for computational efficiency. That is, while $2n$ memory units are required to store n entity identifiers, an entity lookup operation runs in constant time [9]. Both arrays of data are necessary, as each array serves a different purpose. The dense array exists to allow for efficient access and iteration of all entities in the system, while the sparse array serves to allow single entity lookups to operate as efficiently as possible. To look up whether an entity is present in the sparse set, it is a single boolean operation:

```
dense[sparse[entityID]] == entityID
```

If this operation returns $true$, the entity is present in the sparse set structure. If the two vectors do not contain matching information for a certain set of indices, the check will return false and the entity is not in the sparse set.

To add an item to the sparse set, the new item (entity) is pushed back onto the dense array, setting the value of the new item to an empty index in the sparse (an available entity ID). The empty index in the sparse array is set to the index of the entity ID. See Figure 9 for an example.

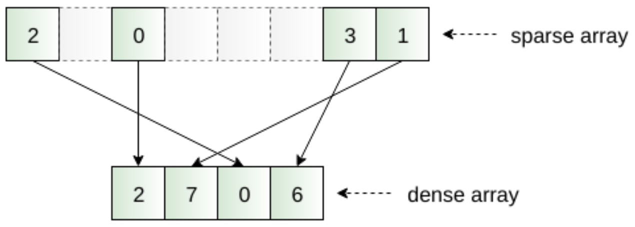

Fig. 9: Diagram of addition of entity to sparse set [8]

Removing an entity from the structure is slightly more complex, but still relatively simple. Using the arrays in Figure 9 as an example, the entity with identifier 6 (index 3 in the dense array) will be removed. In order to remove this entity, it will be swapped with the last entity in the sparse array. In this example, entity 7 is redesignated as entity 6, and $sparse[6]$ is set to 1, the index of the entity ID. The old entity is simply popped off the end of the dense array. Figure 10 shows this removal.

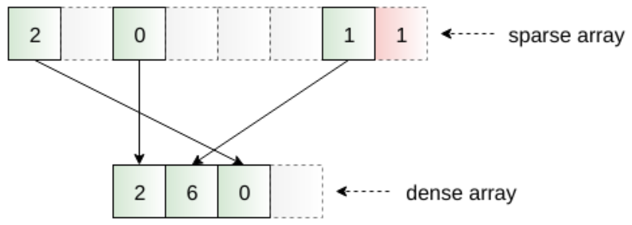

Fig. 10: Diagram of removal of entity from sparse set [8]

Because the entity ID of 7 has been removed from the dense array, a lookup of using entity ID 7 will not pass. Therefore, $sparse[7]$ does not require resetting.

Components are stored in parallel to the dense array of entities, such that the entity at $dense[n]$ has component data of $component[n]$ [12]. Each component type has its own densely packed array. If an entity does not have a certain component, that entity's position in the parallel component array is set to $null$ or another tombstone value that readily indicates that the value at that position is not a component value. Because components are stored parallel to entities, component addition to and removal from an entity is as simple as setting a single value in an array.

A tombstone value can also be helpful when searching for entities in the sparse array. When an entity is removed from the dense array, the location of that entity's identifier is set to the tombstone value [8]. The check for an entity within the sparse set then becomes

```
    sparse[entityID] != tombstone
```

This new look up operation does not require accessing the dense array at all, saving clock cycles in the program.

Given that every entity identifier is stored in the densely packed array contiguously in memory, it is simple and efficient to iterate through all entities within a program. Similarly, as components are arranged into contiguous arrays, it is cache-friendly and efficient to load and iterate all instances of a component [12], [8].

Sparse set organization of data is beneficial to applications that often manipulate the component makeup of entities or need to reference all components of a certain type at one time. However, these programs accept a trade-off. Querying all components for a certain entity or entities is an expensive operation, as each array of components must be loaded and referenced individually [7].

2.3 Structure of Arrays and Array of Structures

Both archetype and sparse set storage frameworks aim to maximize efficiency through the use of vectorization, albeit in different organizations. The two formats of data arrangement can be related to more abstract concepts of data organization - Structure of Arrays (SoA) and Array of Structures (AoS).

A structure of arrays stores all data of the same type in a single array, and stores all relevant arrays in a single struct [13]. An array of structures implements the reverse. A structure is created with all data relevant to that specific structure, and a single array containing repeating storage of that struct contains the data.

A sparse set ECS can be conceptualized at a high-level as an SoA. It groups all data by type and use (component), and a single struct is available to reference the data (the system). Similarly, an archetype ECS can be abstracted as an AoS. Every unique combination of components is a struct (archetype). These structs are then stored together in an array (an archetype table).

It is important to note that the connection between SoA and sparse set, and AoS and archetypal is not one of implementation. The structure of individual components and arrays within both sparse set and archetypal ECSs can be implemented using the developer's choice of memory structure.

3. Survey of ECS Implementations

There are many implementations of ECS-based systems available in both open-source and closed-source formats. While identifying which implementations would be most suitable for this research, eight libraries were examined - Bevy, EnTT, Esper, Flecs, Hecs, Legion, Shipyard, and Specs [14], [15], [16], [17], [18], [19], [20], [21]. Ultimately, three were chosen - Flecs, Bevy, and Esper. Flecs and Bevy were chosen for their thorough documentation and active development communities. The other, Esper, was chosen because of its concise nature and novel implementation. All three libraries analyzed in this research are open-source. Flecs is written in C, Bevy in Rust, and Esper in Python. The choice of different languages for the various libraries in this research was delibrate, attempting to survey as broad a cross-section of the current availability as possible. The survey of these libraries allow for an understanding of ECS across languages, implementations, and uses, providing insight into the actuality of in-memory storage for entity-component systems.

3.1 Flecs

Flecs is a bare-bones, pure-ECS library that facilitates data storage in any variety of applications. However, Flecs advertises itself as useful with "most game engines and scripting languages," indicating some of its common uses [17].

Flecs uses the archetypal system of ECS storage. The developers of Flecs highly value the speed at which code can execute (as do many of Flecs's users), and the densely packed arrays of archetypal storage allow for easier vectorization at compile time [2]. Densely packed arrays for component storage in the archetype table data type indicates that components within a table are stored contiguously in memory. Contiguous arrangement of data allows for query results to be cached in those same densely packed arrays, while also supporting SIMD instructions [2].

Flecs offers a simple querying functionality to access entities and components within an application. When queries are executed, the relevant tables are returned and cached in an internal, Flecs-defined cache structure. Because building query results can be an expensive operation, the internal cache allows for easy reuse of entities without having to rebuild query results. The cache is filled as efficiently as possible using the contiguous arrays, which house the components.

There are two variations of queries in the Flecs library – cached and uncached [17]. Uncached queries (referred to as filters) act as simple iterators to find the queried data and immediately return it. These queries act similarly to a hashmap lookup, and do not store any of the temporary data after it is returned.

The second type of data retrieval mechanism is cached, and is referred to as a Query within the system. Caches are built per query, which adds to the expense of the operation of querying, but once the query is executed, it is simple to iterate the query results whenever necessary.

3.2 Bevy

Bevy is an ECS-based game engine for Rust that emphasizes simplicity for the user. This simplicity is achieved through the use of Rust structs and functions as the means of storing and manipulating data in the program. This research focuses specifically on the Bevy ECS library (crate, in Rust), which can be used in conjunction with the game engine or as a standalone library [14].

Bevy allows the developer to choose what ECS storage paradigm to employ in their program, based on which mechanism benefits their uses [7]. Archetypal ECS queries and iterates entities quickly, but it is known to be expensive to add or remove components to or from entities. Bevy mitigates this drawback by adding directed edges as attributes to the table of each archetype, indicating what archetypes correspond to the addition or subtraction of each possible component [22]. This "archetype graph" that is created can quickly become very complicated. In response, Bevy uses Bundles of components to define edges, where applicable, to make archetype changes en masse. Bevy Bundles can be described as "templates" to quickly create common sets of components simultaneously. Both bundles and components are simple Rust struct data types, however, which can lead to potential bugs remaining uncaught by the compiler. Part of Rust's usefulness is based off of its ability to catch runtime problems that other languages may miss, so this fact is less than ideal. However, tuples of components are also considered by Bevy to be Bundles, which allows for compiler differentiation of data types, but less readability and versatility for the developer.

The data in Bevy applications that use sparse set storage structures are stored by component in Rust vector data types (Vec). Using data types native to the language allows components to be packed as densely and efficiently as possible. The component data struct is defined in the library; information about each specific component is stored contiguously in the component. Data in archetype storage based applications is stored by column, similar to sparse set storage. Each column in an archetype is the same component type, so each column is stored as a BlobVec (a type-erased vector type), packed as densely as possible to make the most efficient use of memory space. To create links between the columns to define each archetype table, the sparse set data type is used to create and store references to all the columns within a table.

Bevy uses a stateful querying system as a means to cache query results. When a query is executed, the resulting archetype is stored in the query state. This stateful mechanism means that the expensive operations within a query, namely locating matching components and entities, only need to be executed when the query is first constructed. Waiting until a query is first built to execute its expensive operations also allows for iterative expansion of the state, as the addition of new archetypes does not require re-execution of previously queried archetypes.

3.3 Esper

Esper is an open-source Python 3 library for entity-component systems. It was developed with performance in mind, specifically focused on cache efficiency. Esper differs greatly from the other ECS libraries in this research because it does not specifically use archetypal or sparse set storage paradigms. Esper uses its own sparse-set-adjacent storage system, not giving much thought to where data is stored in memory. Instead, the focus of data organization is placed on how it occurs within the cache.

The Esper library utilizes a Python functools functionality known as the least-recently-used (LRU) cache [16]. The LRU cache allows Esper to operate fundamentally differently from other ECS implementations. While other implementations employ component archetypes to organize data efficiently in memory before it is ever queried, Esper gives no thought to in-memory storage. Instead, it utilizes Python's native List data structure to organize components in memory. Python Lists act as linked lists, maintaining pointers to the items in memory [23], [24]. Pointers are stored contiguously in memory, and Lists are dynamically sized to support as many items as needed. Because Esper uses Python Lists to store components non-contiguously in memory, it leverages the LRU cache to organize components contiguously in the cache for delivery to the program and/or user. Components stay in the cache, and therefore organized, until the cache is full, and then cached queries are overwritten, with the least-recently used queried overwritten first.

This system does have limitations. Since query results are organized in the cache, as opposed to in-memory, if the data in a component is changed, the cache is invalidated and the query must be re-executed to retrieve the most recent data. If component data is changed often, constant re-rerunning of queries can quickly become computationally expensive.

4. Conclusions

The main goal of this research is to find what, if any, overarching in-memory data storage scheme was pervasive in entity-component system libraries. Through the survey of the three representative libraries examined in this research, it is indicated that the basic framework for ECS storage is that of densely packed arrays. For the purposes of this discussion, focus is placed on Flecs and Bevy, while Esper is placed in its own category, given the novelty of its approach.

The following table, table 1, summarizes the similarities and differences between the libraries surveyed in this research.

	Flecs	Bevy	Esper
Language	C	Rust	Python
Supports Archetypal Storage	Yes	Yes	No
Supports Sparse Set Storage	No	Yes	No
Uses Internal Cache	Yes	Yes	Yes
Graphs Archetype Tables	Yes	Yes	No

Table 1: Summary of ECS Libraries

In both Flecs and Bevy, archetypal storage is organized by column – the data for each table is stored by grouping like components together in dense arrays. Columns of components are stored contiguously in memory. Bevy and Flecs differ on whether they gather columns contiguously in memory. Bevy uses its internal sparse set data structure to organize references to each column. Flecs organizes all columns of a table contiguously in memory. The contiguous organization of data in memory also means that the application's query cache can be filled as efficiently as possible. Contiguous data also supports the use of SIMD instructions to be executed on data in the program.

Flecs and Bevy both employ a table graphing system. This graphing system creates connections between archetype tables, depending on what components make up each archetype. In the graph, each individual archetype is a node, and each relevant manipulation (i.e. addition or removal) of one or more components is an edge to another node. If there is not a corresponding archetype for an edge to connect to, the edge does not exist. The graphing system allows entities to be transferred between archetypes as efficiently as possible. If a graph of existing archetypes did not exist, every individual archetype table would need to be iterated when components are added or removed from an entity. Instead, the component or components that are added or removed can be quickly found on the edges from the archetype where the entity is currently stored, and the new archetype is found.

In contrast to Bevy and Flecs, Esper gives no thought to how data is organized and stored in memory, therefore it does not fit the mold of the more traditional ECS implementations. Components and entities are not organized into sparse sets or archetypes. This results in component and entity data not being linked through any sort of graph system. However, Esper, like Bevy and Flecs, does use an internal caching system to quickly reference queries.

Because the action of searching for and retrieving data from program storage can be computationally expensive, relative to the size and amount of data being retrieved, all libraries examined in this research employ an internal caching system to retain query results for later use. Caching queries is also beneficial when the number of archetypes grows. Just as retrieving more data requires more effort on the part of the application, as more data is introduced, creating more information through which to search, more effort is required. When queries only have to be executed once, and the results can be cached to be used again, the system can operate with greater efficiency. However, caching queries is not as beneficial in scenarios where components are constantly changing, since the cache would continually be invalidated.

The information presented in this research aims to specifically document the in-memory organization of common ECS storage frameworks. The use of ECS in data-intensive applications can serve to benefit computational efficiency. As data processing needs continue to grow in all sectors of software applications, the need for efficiency grows in parallel. In-depth understanding and documentation further improve all future implementations and uses of entity-component systems.

5. Acknowledgements

We would extend our gratitude to Sander Mertens and Michele Caini for their continuing work on and documentation of ECSs. Their in-depth explanations of many aspects of entity-component systems served as invaluable references in this research.

6. Disclaimer

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the U.S. Government.

References

[1] R. Fabian, *Data-Oriented Design*, C. Ring, Ed., 2018. [Online]. Available: www.dataorienteddesign.com
[2] S. Mertens, "ECS FAQ," 2021. [Online]. Available: https://github.com/SanderMertens/ecs-faq
[3] "ECS concepts." [Online]. Available: https://tinyurl.com/yckrhhfd
[4] J. Vagedes, "A study of execution performance for rust-based object vs data-oriented architectures," Ph.D. dissertation, Air Force Institute of Technology, 2020.
[5] S. Mertens, "Building an ECS 2: Archetypes and Vectorization," 2020. [Online]. Available: https://ajmmertens.medium.com/building-an-ecs-2-archetypes-and-vectorization-fe21690805f9
[6] M. Caini, "ECS back and forth - Part 2," 2019. [Online]. Available: https://skypjack.github.io/2019-03-07-ecs-baf-part-2/
[7] C. Anderson, "Bevy 0.5," 2021. [Online]. Available: https://bevyengine.org/news/bevy-0-5/
[8] M. Caini, "ECS back and forth - Part 9," 2020. [Online]. Available: https://skypjack.github.io/2020-08-02-ecs-baf-part-9/
[9] "Sparse Sets," 2012. [Online]. Available: https://programmingpraxis.com/2012/03/09/sparse-sets/
[10] [Online]. Available: https://leatherbee.org/index.php/2019/09/12/ecs-1-inheritance-vs-composition-and-ecs-background/
[11] H. Stefan, "Adventures in data-oriented design – Part 3c: External References," 2013. [Online]. Available: https://blog.molecular-matters.com/2013/07/24/adventures-in-data-oriented-design-part-3c-external-references/
[12] D. Komer, "Ecs with sparse array notes (entt style)," 2020. [Online]. Available: https://tinyurl.com/2p922r6a
[13] J. McMurray, "A introduction to Data Oriented Design with Rust," 2020. [Online]. Available: jamesmcm.github.io/blog/2020/07/25/intro-dod/
[14] C. Anderson, "Bevy," 2022. [Online]. Available: https://github.com/bevyengine/bevy
[15] M. Caini, "Entt," 2022. [Online]. Available: https://github.com/skypjack/entt
[16] B. Moran, "Esper," 2021. [Online]. Available: https://github.com/benmoran56/esper
[17] S. Mertens, "Flecs," 2022. [Online]. Available: https://github.com/SanderMertens/flecs
[18] B. Saunders, "Hecs," 2022. [Online]. Available: https://github.com/Ralith/hecs
[19] "Legion," 2022. [Online]. Available: https://github.com/amethyst/legion
[20] "Shipyard User's Guide." [Online]. Available: https://leudz.github.io/shipyard/guide/master/welcome.html
[21] "Specs Documentation," 2021. [Online]. Available: https://docs.rs/specs/0.17.0/specs/storage/index.html
[22] I. Iyes, "The Unofficial Bevy Cheat Book," 2022. [Online]. Available: https://bevy-cheatbook.github.io/introduction.html
[23] "How is Python's List Implemented?" 2014. [Online]. Available: https://stackoverflow.com/questions/3917574/how-is-pythons-list-implemented
[24] "How are Python lists implemented internally ?" 2013. [Online]. Available: https://www.quora.com/How-are-Python-lists-implemented-internally
[25] M. Caini, "ECS back and forth - Part 1," 2019. [Online]. Available: https://skypjack.github.io/2019-02-14-ecs-baf-part-1/
[26] S. Mertens, "Flecs (not for dummies)," 2021. [Online]. Available: https://github.com/SanderMertens/flecs_not_for_dummies
[27] The Cherno, "Entity Component System | Game Engine series," 2020. [Online]. Available: https://www.youtube.com/watch?v=Z-CILn2w9K0
[28] P. Briggs and L. Torczon, "An Efficient Representation for Sparse Sets," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 2, no. 1-4, pp. 59–69, 1993.
[29] M. Acton, "Data-Oriented Design and C++," in *Cpp-Conference*, 2014. [Online]. Available: https://www.youtube.com/watch?v=rX0ItVEVjHc
[30] M. Kleppmann, *Designing Applications Data-Intensive Application*, A. Spencer and M. Beaugureau, Eds. O'Reilly Media, Inc., 2017.
[31] T. Neward, "The Vietnam of Computer Science," 2006. [Online]. Available: http://blogs.tedneward.com/post/the-vietnam-of-computer-science/
[32] S. Mertens, "Designing with Flecs," 2021. [Online]. Available: https://tinyurl.com/v4kbs638
[33] Mothfuzz, "Archetypes," 2022. [Online]. Available: https://github.com/mothfuzz/Archetypes
[34] S. Gupta, "Caching in Python: the LRU algorithm," 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/08/caching-in-python-the-lru-algorithm/
[35] C. Anderson, "Bevy ECS V2 Merge Request 1525," 2021. [Online]. Available: https://github.com/bevyengine/bevy/pull/1525
[36] "ECS Deep Dive," 2019. [Online]. Available: https://rams3s.github.io/blog/2019-01-09-ecs-deep-dive/
[37] T. Gillen, "Archetypal vs Grouped ECS Architectures, my take," 2020. [Online]. Available: https://community.amethyst.rs/t/archetypal-vs-grouped-ecs-architectures-my-take/1344
[38] I. Kettlewell, "Writing a tiny Entity Component System in Rust," 2021. [Online]. Available: https://ianjk.com/ecs-in-rust/
[39] "Managing Decoupling Part 4 – The ID Lookup Table," 2011. [Online]. Available: http://bitsquid.blogspot.com/2011/09/managing-decoupling-part-4-id-lookup.html
[40] S. Meyers, "CPU Caches and Why You Care." Oslo, Norway: NDC Conference, 2014. [Online]. Available: https://vimeo.com/97337258

Author Biographies

BAILEY V. COMPTON is a 2nd Lieutenant currently stationed at Wright Patterson Air Force Base, Ohio. She is a Master's Student at the Air Force Institute of Technology (AFIT) studying Computer Science. Lt Compton is a 2020 graduate of the United States Air Force Academy, where she studied Computer Science.

MAJOR RICHARD DILL is an Assistant Professor of Computer Science at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Computer Science from the University of Maryland at College Park in 2004, and both an M.S. in Computer Science in 2008 and a Ph.D. in 2018 from AFIT. Major Dill's research interests include computer security, algorithms, and artificial intelligence.

DOUGLAS D. HODSON is an Associate Professor of Computer Engineering at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Physics from Wright State University in 1985, and both an M.S. in Electro-Optics in 1987 and an M.B.A. in 1999 from the University of Dayton. He completed his Ph.D. at the AFIT in 2009. His research interests include computer engineering, software engineering, real-time distributed simulation, and quantum communications. He is also a DAGSI scholar and a member of Tau Beta Pi.

MICHAEL R. GRIMAILA (BSEE 1993; MSEE 1995; Ph.D. 1999, Texas A&M University) is a professor and head of the Department of Systems Engineering and Management at the Air Force Institute of Technology, Wright-Patterson Air Force Base in Ohio, USA. He is a member of Tau Beta Pi, Eta Kappa Nu, and the Association for Computing Machinery, as well as a Senior Member of the IEEE, and a Fellow of the Information System Security Association. He can be contacted via email at michael.grimaila@afit.edu.

IV. Conclusions

4.1 Research Summary

This thesis set out with three research goals. The first goal, that of documentation of the common storage models used by entity-component systems, was covered in chapter II and chapter III. Both of these sections discuss the organization of data within an ECS, specifically in archetype and sparse set based systems. Chapter I introduce the concepts of Structure of Arrays (SoA) and Array of Structures (AoS). SoA and AoS are simply ways to organize data, just as archetypes and sparse sets are. In fact, the connection between SoA, AoS, and ECS organizations is important. AoS and SoA are well-defined and pervasive concepts within computer science, and making a connection between those ideas and the organization of ECS creates greater understanding of ECS. However, as mentioned in chapter III, these are conceptual connections at an abstract level. In implementation, both archetypes and sparse sets can use either AoS or SoA in their memory organizations.

Chapter II discusses at length the comparisons to be made between relational databases and entity-component systems, fulfulling another research goal. Database normalization is a well-documented topic with standards and rules. These rules and standards can be applied to the atomization of components within an ECS. Accessing more than 50 years of previous research and documentation on the topic can only be beneficial to the ECS field.

The third topic of note in this research is an in-depth understanding of the organization of data in program memory of an ECS application. This is satisfied in chapter III.

The information contained in this thesis provides direct benefits to the United States Air Force (USAF) and Department of Defense (DoD), as ECS systems are applicable to many training and defense systems. Data, and the organization and use thereof, affects the realms of air, space, and cyberspace, and will only grow in influence in the future.

4.2 Future Work

In the future, the research introduced in this thesis can be continued through several avenues. Three such lines of effort are outlined here.

The first suggested future work may investigate the feasibility of a third type of ECS storage framework that is able to leverage both sparse set and archetype storage frameworks simultaneously. This combination framework would possibly be able to mitigate the inefficiencies in the archetype and sparse set frameworks, and allow for even more optimized program execution.

Another area of future research on this topic relates to the use of graphs within archetype storage organization. Many problems within NP-space relate to graphs, such as the set cover problem. Research into the application and approximation of solutions to these problems within an archetypal ECS would be beneficial to future ECS implementations.

The third avenue of research pertains to the relationship between databases and ECS systems. Expanding on the research in this thesis, database types other than relational, such as NoSQL and columnar, can be added to the comparison to widen the basis of research and potential processing benefits inherent to these methods.

Bibliography

1. M. Geldard, T. Nugent, and P. Buttfield-Addison, "Software architecture conference proceedings: 'entity component systems and you: They're not just for game developers'," New York, NY, 2020. [Online]. Available: https://conferences.oreilly.com/software-architecture/sa-ny/public/schedule/detail/79975.html

2. N. Llopis, "Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP)," 2009. [Online]. Available: http://gamesfromwithin.com/DATA-ORIENTED-DESIGN

3. R. Fabian, *Data-Oriented Design*, C. Ring, Ed., 2018. [Online]. Available: www.dataorienteddesign.com

4. United States Government, *National Defense Strategy.* Unites States Government, 2018. [Online]. Available: https://dod.defense.gov/Portals/1/Documents/pubs/2018-National-Defense-Strategy-Summary.pdf

5. C. Anderson, "Bevy," 2022. [Online]. Available: https://github.com/bevyengine/bevy

6. M. Caini, "Entt," 2022. [Online]. Available: https://github.com/skypjack/entt

7. B. Moran, "Esper," 2021. [Online]. Available: https://github.com/benmoran56/esper

8. S. Mertens, "Flecs," 2022. [Online]. Available: https://github.com/SanderMertens/flecs

9. "Legion," 2022. [Online]. Available: https://github.com/amethyst/legion

10. B. Saunders, "Hecs," 2022. [Online]. Available: https://github.com/Ralith/hecs

11. D. Ancel, “Shipyard,” 2022.

12. “Shipyard User’s Guide.” [Online]. Available: https://leudz.github.io/shipyard/guide/master/welcome.html

13. “Specs Documentation,” 2021. [Online]. Available: https://docs.rs/specs/0.17.0/specs/storage/index.html

14. S. Mertens, “Building an ECS 2: Archetypes and Vectorization,” 2020. [Online]. Available: https://ajmmertens.medium.com/building-an-ecs-2-archetypes-and-vectorization-fe21690805f9

15. “Array of Structures vs . Array within a Structure,” 2020. [Online]. Available: https://www.geeksforgeeks.org/array-of-structures-vs-array-within-a-structure-in-c-and-cpp/

16. J. Pennycook, S. Hammond, S. Wright, A. Herdman, I. Miller, and S. Jarvis, “An investigation of the performance portability of opencl,” *Journal of Parallel and Distributed Computing*, vol. 73, p. 1439–1450, 11 2013.

Acronyms

AoS Array of Structures. 6, 7, 24

API Application-Programmer Interface. 4

C4ISR Command, Control, Communications, Computers and Intelligence, Surveillance, and Reconnaissance. 3

DoD Department of Defense. 3, 24

ECS Entity-Component Systems. iv, 2

ECS Entity-Component System. 3

NDS National Defense Strategy. 3

SIMD Single Instruction, Multiple Data. 6

SoA Structure of Arrays. 6, 24

USAF United States Air Force. 24

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)* 24–03–2022	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Sept 2020 — Mar 2022

4. TITLE AND SUBTITLE An Investigation of Data Storage in Entity-Component Systems	**5a. CONTRACT NUMBER** **5b. GRANT NUMBER** **5c. PROGRAM ELEMENT NUMBER**
6. AUTHOR(S) Bailey V. Compton	**5d. PROJECT NUMBER** **5e. TASK NUMBER** **5f. WORK UNIT NUMBER**
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENG-MS-22-M-018
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RQQD Building 145 WPAFB OH 45433-7765 DSN 798-6556, COMM 937-904-6556 Email: James.Zeh@us.af.mil	**10. SPONSOR/MONITOR'S ACRONYM(S)** **11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Entity-Component Systems (ECS) have grown vastly in application since their introduction more than 20 years ago. Providing the ability to efficiently manage data and optimize program execution, ECSs, as well as the wider field of data-oriented design, have attained popularity in the realms of modeling, simulation, and gaming. This manuscript aims to elucidate and document the storage frameworks commonly found in ECSs, as well as suggesting conceptual connections between ECSs and relational databases. This formal documentation of the in-memory storage formats of entity-component systems affords the United States Air Force, the Department of Defense, and the software engineering community a greater understanding and applicability of the paradigm.

15. SUBJECT TERMS

entity-component system, ECS, data-oriented design, dod, relational database, archetype, sparse set, structure of arrays, array of structures

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Douglas D. Hodson, AFIT/ENG
U	U	U	UU	37	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-3636, ext 4719