Entity Component Systems: An Alternative to SQL?

Robert Burnett
Computer Science Department
Angelo State University
San Angelo, United States
rburnett5@angelo.edu

***Abstract*—An Entity Component System (ECS) is a commonly used data structure in game development that assigns independent data units, called components, to unique identifiers known as entities. This arrangement allows specialized functions called systems to efficiently query these entities based on the components they possess. This research explores repurposing an ECS as an alternative to Structured Query Language (SQL) for general-purpose databases, with a focus on business management systems. By drawing parallels between ECS entity lookups and SQL database queries, this study investigates whether the inherent advantages of an ECS-based approach—such as superior performance, optimized hardware utilization, and reduced dependency on complex domain-specific languages—can address the limitations of traditional SQL databases for certain use cases. A prototype implementation serves as the basis for benchmarking and trade-off analysis, offering insights into the potential benefits and drawbacks of adopting ECS architectures outside the game development sphere.**

***Index Terms*—ECS, Database, SQL**

I. Challenges / Constraints

Multiple important technological considerations constrain the design of an ECS focused database system. First, because one of the objectives of this research is to remove dependencies on complex domain specific languages, the programming language that such an application is developed in must be sufficiently expressive enough to allow for ergonomic queries of the database using that languages built in syntax. Existing SQL databases generally require SQL language queries to be passed as strings into database library functions. This approach has multiple downsides. Structuring database queries as raw strings of text doesn't allow for much sophisticated compile-time checking to occur, and makes syntax highlighting difficult. Thus, the chosen language must be sufficiently expressive.

Next, and very importantly, the language must support some degree of reflection. Reflection is the ability of a programming language to examine its own source code, either at runtime or compile-time. Reflection makes the task of serialization (ie, converting in-memory data structures into a format that can be saved on disk and reloaded later) much easier. In languages which do not support reflection, the code for serializing any given arbitrary data structure must be written out manually, since any serialization code is not able to reflect on how the data is structured in memory. This violates the principle of DRY (Do not Repeat Yourself), since the structure of data in memory must be specified in multiple places. First: where that structure is initially defined. And second: in the logic of where that structure is serialized and deserialized. In a language with reflection, serialization code can reflection on structures directly.

II. Methodology

To test the viability of an ECS for building non-game related projects, work began on an experimental database prototype. First, a language for this prototype had to be chosen. Because of the aforementioned constraints of expressiveness and serialization, Common Lisp was chosen as the language for the prototype.

Beyond these constraints, Common Lisp also has several other benefits that gave it an edge over the other languages considered. First, the ecosystem of Common Lisp is very stable. Development on the language first began in 1981 (White, J), and the language was standardized by ANSI in 1994 [1]. Several other languages considered are still under active development, and are still very unstable. Second, Common Lisp has the advantage of fast optimizing compilers such as SBCL (Carnegie Mellon University, et al). This allows Common Lisp to have competitive performance against other languages.

The ECS architecture chosen for the prototype was one based on sparse sets. A sparse set is a data structure that emulates a large array of elements but without needing as much memory as a large array, provided the set is populated sparsely. Every type of component in the ECS is then stored in its own sparse set, and the ids of entities which are represented by a large integer are used to index into the sparse set. Given that every entity will not have every type of component, this is a much more memory efficient way of storing components than by storing each type of component in a large array, because most of the indexes in that array will be empty and unused.

In order to emulate the ability of SQL to query database tables, a system was developed to query the ECS for entities that have a matching set of components. This was done by storing a list of entity ids for every type of component. Whenever an entity was given a particular component, that entity's id would be added to the list of entities that had that type of component. Then, querying for a list of entities that have a given component is possible by taking the intersection of the entity id list for each desired component. For example, in our hypothetical database let us say we have the following component types.

- Address: String - Phone Number: String - Email: String - Blacklisted: BitFlag - Business Name: String

We can easily query for all entities which have the Business Name and Blacklisted components, and thereby access other information about businesses which are blacklisted.

(print-all-components *ecs* (find-entities *ecs* :business-name :blacklisted))

III. Current Findings

Throughout building the prototype, several key challenges inherent to a ECS architecture were identified.

First, an ECS relies on the representation of data in memory in order to perform efficient queries. It is not immediately obvious how the relationship between in memory data and on disk data should be managed. It is important for data to be backed up on disk. However, performing disk IO operations for every small query could have negative performance consequences.

Another challenge relates to the optimal manner of storing serialized ECS data on disk. Incremental updates to data stored on disk will be more efficient than backing up the entire ECS for every update. This means care must be taken to devise a system for storing data that allows for specific sections to be isolated and updated independently.

Furthermore, in business critical systems, the method of storing data must be amenable to keeping backups and changelogs of how data was changed and when. Being able to revert the database to an older state is important in the case of mistakes or accounting errors.

One possible solution to these problems uncovered in this initial research is to utilize source control programs to manage database state. Source control programs are a category of software used to manage source code by storing incremental changes to a codebase. At any point, the source code can be reverted to an older version. Knowing this, one hypothetical solution to the data integrity problem would be to serialize data about each unique entity in a file named by that entity's id. That file can then be queried and updated whenever an entity is modified. Then, changes to entity data files across the system can be committed into source control, allowing for the database to be easily backed up and reverted if necessary.

IV. Future Work

Future work on this project will involve research into several problems uncovered by the initial prototype. The first new pursuit will be work on integrating a graphical user interface (GUI) into the program, which will allow end users to easily query the database, create new components, and perform other similar tasks.

Second, regarding the need to backup older editions of the data, efforts will be made to integrate source control software into the system. The most likely candidate for this purpose will be Git, which is the most widespread source control software in the present day. Git is very stable, and is used for many very important codebases, such as the Linux Kernel (Torvalds, et al).

Git is also a distributed version control system. One interesting possibility this raises relates to the usage of Git to create a centralized database hosted on one of the many secure and free Git hosting platforms.

References

[1] ANSI, "226-1994 (r1999)," Tech. Rep., 1994.