README.md for SGS project

This document provides a technical overview of the development process for the SGS project. For a more in-depth understanding of the project's evolution, we recommend reviewing the Introduction.md file.

Attention: This is a work in progress, with new chapters and editing to come.

Setting up development environment

I have selected Visual Studio Code as my preferred IDE for multiple reasons. It provides strong support for Julia, with the Jupyter Extension being particularly well-equipped for managing Julia code. Personally, I find the use of notebooks as a testing environment to be extremely convenient. However, feel free to choose any IDE that best fits your preferences.

Project include:

- **SGS as a Julia framework**: Streamlining operations and maintaining our commitment to an embeddable framework.
- **Redis**: Serving as a computational engine, database, and search engine, **Redis** complements the **SGS** framework like a trusted tool—indispensable but unobtrusive, much like a cellphone that, while not a part of you, is always by your side.

Installing Julia

The most recommended method for installing Julia is by utilizing the official Julia packaging. As of now, I am running Julia version 1.10.4.

Installing Redis

I am utilizing the official Redis Docker image. For detailed guidance on how to use this image, please refer to the following article: [How to Use the Redis Docker Official Image](https://www.docker.com/blog/how-to-use-the-redis-docker-official-image/).

To install it, execute the following Bash command:

```
Unset
podman run -d --name redis-stack -p 6379:6379 -p 8001:8001 latest
```

This will install the Redis server and RedisInsight on port 8001. Once installed, you can access RedisInsight from your browser using the following link: http://localhost:8001/.

If you prefer using the REPL interface, you will need to install the Redis server locally, which includes the redis-cli tool for starting the REPL.

Cloning SGS project

```
Unset
git clone https://github.com/alexmy21/SGS.git
```

Open project in VS Code

```
Unset
cd SGS
code .
```

When you launch VS Code, open the terminal window and execute the following command:

```
Unset
julia> using Pkg
julia> Pkg.activate(".")
```

SGS Architecture

The core architecture of SGS is encapsulated within eight essential files:

1. **constants.jl:** This file contains a collection of constants critical for the HyperLogLog algorithm used in sets.jl.
2. **sets.jl:** This script implements HllSets, which serve as fundamental building blocks for all data structures within SGS.
3. **entity.jl:** This module introduces a new structure called Entity, which encapsulates metadata using HllSets. This initiative builds on Mike Saint-Antoine's work with SimpleGrad.jl, adapted from Andrey Karpathy's MicroGrad project. Unlike the original frameworks that utilize numerical embeddings in their neural networks, our approach substitutes these embeddings with HllSets (see [1, 2, 3, 4]).

4. **graph.jl:** This module implements the Graph structure, employing wrapped HllSets as nodes and representing edges as pairs of connected nodes. It also supports all set operations on nodes, adapted from the corresponding operations in HllSets (sets.jl).
5. **store.jl:** Serving as the data management hub, this module oversees all data-related operations, including data ingestion, processing scheduling, commits, and import/export functions. Essentially, store.jl facilitates all operations and tools necessary to support the SGS self-generative loop.
6. **search.jl:** This file provides support for search operations on Redisearch indices.
7. **tokens.jl:** This module is dedicated to managing token inverted indexes, linking datasets with their content treated as collections of tokens. All unique tokens extracted from the datasets are stored within Redisearch token indices.
8. **utils.jl:** A collection of general utility functions that are common across all files.

Among these, store.jl stands out for performing the most complex processing by leveraging functions from other modules.

The strength of SGS lies in its meticulously engineered data architecture, specifically designed and optimized to meet the unique requirements of a self-reproducing loop.

Metadata as the Foundation of SGS

There is a common perspective that numerical embeddings in standard neural network models are created by transforming tangible entities into numerical formats. While this approach effectively represents elements of reality, our focus shifts toward employing metadata to depict these elements in a more abstract manner. One of the key benefits of metadata is its capacity to organize elements into groups that share semantic similarities, although these groups are not completely distinct and often exhibit overlapping characteristics.

Adopting metadata signifies a fundamental shift, profoundly altering the way AI models are developed. Consider the difference between training dogs and educating humans: both processes aim to instill an understanding of causal relationships, but the complexity of the logic varies significantly. Dog training relies on simple cause-and-effect logic, where sequential actions are interpreted as having causal relationships, but human education embraces a more intricate logic where cause and effect are not necessarily directly linked. Presently, our AI training techniques resemble methods more suitable for dog training.

By abstracting real-world entities into metadata, we enhance the logic used, moving beyond the simplistic cause-and-effect, or "dog logic." In the domain of metadata, the principle of "this follows that" more reliably signals a causal relationship, often implying a "because of that" rationale.

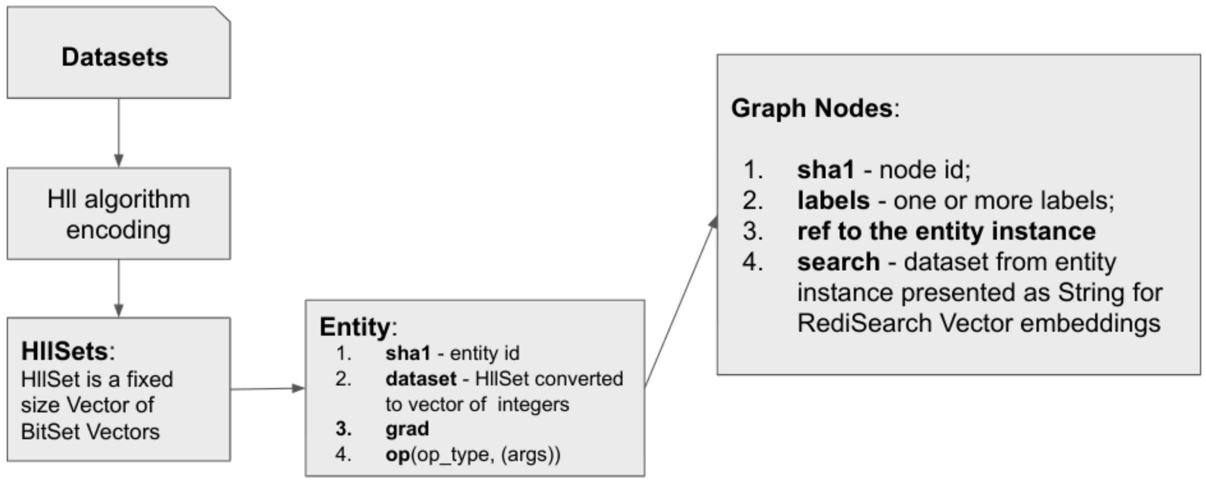
A critical aspect of our approach is that each metadata item correlates with a specific group of elements. Within the metadata domain, an HllSet acts as an embedding for these groups. Unlike traditional numerical representations, this embedding takes the form of a fixed-size bit-vector, specifically a 2-dimensional Tensor (64, P). In Julia programming language, this is represented as Vector{BitVector}, where each vector is 64 bits in length, and the number of these bit-sets is defined by the parameter P, which determines the precision of the HyperLogLog approximation for the entity collection.

HllSets provide a versatile method for comparing different instances across a wide range of metrics, effectively fulfilling the primary objective of embedding. Moreover, HllSets offer additional advantages, proving to be a dependable alternative to original datasets and fully supporting all set operations.

SGS data structure

Building blocks

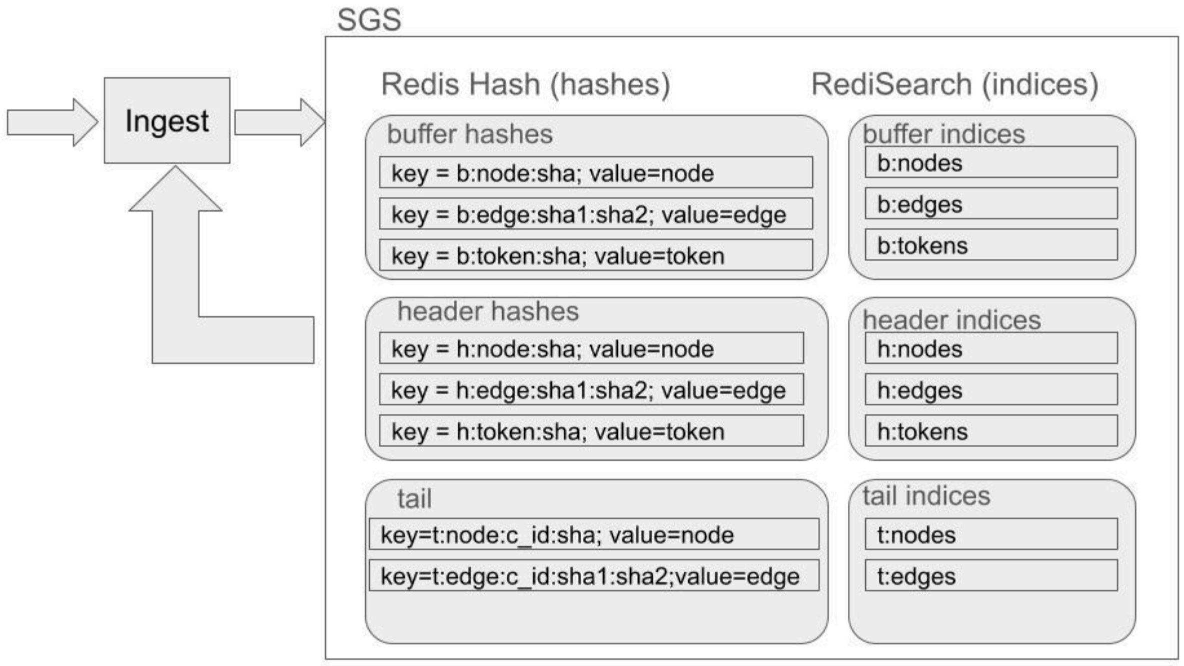
The diagram provided below depicts the relationships among datasets, HllSets, and Graph Nodes.


Datasets
Hll algorithm encoding
HllSets:
HllSet is a fixed size Vector of BitSet Vectors
Entity:
1. sha1 - entity id
2. dataset - HllSet converted to vector of integers
3. grad
4. op(op_type, (args))
Graph Nodes:
1. sha1 - node id;
2. labels - one or more labels;
3. ref to the entity instance
4. search - dataset from entity instance presented as String for RediSearch Vector embeddings

It is crucial to note that there is a seamless transformation from an HllSet to a dataset within a Graph Node, and vice versa.

Data Structure. General overview

This illustration serves as an informative infographic of the SGS data architecture.


SGS
Ingest
Redis Hash (hashes)
RediSearch (indices)
buffer hashes
key = b:node:sha; value=node
key = b:edge:sha1:sha2; value=edge
key = b:token:sha; value=token
buffer indices
b:nodes
b:edges
b:tokens
header hashes
key = h:node:sha; value=node
key = h:edge:sha1:sha2; value=edge
key = h:token:sha; value=token
header indices
h:nodes
h:edges
h:tokens
tail
key=t:node:c_id:sha; value=node
key=t:edge:c_id:sha1:sha2;value=edge
tail indices
t:nodes
t:edges

Examining the diagram might easily give the mistaken impression that we are dealing with at least three distinct types of hashes. However, structurally, all hashes are identical; we categorize them into three groups based on Redis's naming conventions for hash names. Each hash name supports a compound structure consisting of multiple parts divided by a colon (:), referred to as prefixes.

The initial letter in the hash key (b for buffer, h for header, t for tail) denotes the life cycle stage of the specified node instance. As it transitions from one stage to the next, the node itself remains unchanged. In the SGS, while the names may alter, the values remain consistent.

Redis as an integral part of SGS

Redis was selected as the data management tool for SGS following a thorough vetting process. About 10 years ago, we began incorporating Redis into our Java-based projects. Within Java environments, Redis proved effective for managing runtime data. However, interfacing with Redis from Java presented challenges, particularly when utilizing custom-built Redis extension modules. Redis naturally aligns more closely with Python environments, facilitating its integration with Julia and thus making it a suitable runtime data management tool for Julia as well.

RediSearch, an extension of Redis, utilizes the naming convention feature of Redis to link search indices with hashes. For example, the index 'b:nodes' is connected through a schema to the hash prefix 'b:node'. RediSearch tracks all new hashes and associates those starting with a specified prefix in any index to the corresponding index.

A notable feature of Redis is the RENAME command, which allows users to change the name of any key, including hash keys. By renaming a hash from 'b:node:some_sha1_id' to 'h:node:some_sha1_id', we can shift this hash from the 'b:nodes' index to the 'h:nodes' index without physically relocating the data, exemplifying the "zero copy" concept.

As detailed in the Introduction.md document, the data acquisition process in SGS involves buffering all newly acquired data in the transaction stage before committing it to the head. If the head already contains data slated for commitment, the commit module first transfers the existing dataset to the tail by renaming it—another "zero copy" maneuver—before placing the new dataset into the head, also via renaming.

Redis provides essential tools to support the Self Reproducing loop in SGS. We have already highlighted the RENAME command and Redis hashes that aid in representing and transforming HllSets. A critical feature for SGS in Redis is the built-in key's TTL (Time To Live) and eviction mechanism.

Redis TTL offers several key benefits:

- **Cache Management**: It is extensively used to manage the lifespan of data in caches, allowing for the automatic removal of expired data and ensuring that caches remain relevant and current.
- **Session Management**: In web applications, Redis TTL helps manage the duration of user sessions, automatically clearing inactive sessions to free up resources.
- **Scheduled Tasks**: TTL can be employed to time the cessation or refreshment of tasks, such as email dispatches or data updates.
- **Clean-up of Old Data**: TTL facilitates the automatic purge of outdated, irrelevant data, reducing the need for manual intervention.

Moreover, Redis supports eviction policy management, which is crucial for managing cache size and memory usage. As the cache reaches capacity, a vital decision must be made: whether to reject new data or create space by discarding older data.

SGS is fundamentally immutable, only altering certain parameters like gradient rather than its building elements. Instead, it generates new instances of these elements through the reproduction process. This leads to the exponential accumulation of obsolete elements that need to be removed under specific conditions, a task at which Redis excels.

Testing SGS modules

Module: sets.jl

Below are excerpts from the hll_sets.ipynb Jupyter notebook.
In the initial cell, we are configuring the environment to execute the upcoming code:

```
Python
using Random
using FilePathsBase: extension, Path
include("src/sets.jl")

import .HllSets as set
```

Let's generate a few HllSets to experiment with.

```
Python
# Initialize test HllSets
hll1 = set.HllSet{10}()
hll2 = set.HllSet{10}()
hll3 = set.HllSet{10}()
hll4 = set.HllSet{10}()
hll5 = set.HllSet{10}()

# Generate datasets from random strings
s1 = Set(randstring(7) for _ in 1:10)
s2 = Set(randstring(7) for _ in 1:15)
s3 = Set(randstring(7) for _ in 1:100)
s4 = Set(randstring(7) for _ in 1:20)
s5 = Set(randstring(7) for _ in 1:130)

# Add datasets to HllSets
set.add!(hll1, s1)
set.add!(hll2, s2)
set.add!(hll3, s3)
set.add!(hll4, s4)
set.add!(hll5, s5)
```

The following code will demonstrate the effectiveness of HyperLogLog Set approximation in accurately estimating set cardinality:

```
Python
# Print cardinality of datasets and HllSets side by side
print(length(s1), " : ", count(hll1), "\n")
print(length(s2), " : ", count(hll2), "\n")
print(length(s3), " : ", count(hll3), "\n")
print(length(s4), " : ", count(hll4), "\n")
print(length(s5), " : ", count(hll5), "\n\n")
# union
print(length(s1 ∪ s2 ∪ s3 ∪ s4 ∪ s5), " : ", count(hll1 ∪ hll2 ∪ hll3 ∪
hll4 ∪ hll5), "\n")
# intersection
print(length(s1 ∩ s2 ∩ s3 ∩ s4 ∩ s5), " : ", count(hll1 ∩ hll2 ∩ hll3 ∩ hll4 ∩
hll5), "\n")
```

Here is an output demonstrating that the approximation using HllSet is indeed quite effective:

```python
10 : 10
15 : 15
100 : 98
20 : 19
130 : 129

275 : 269
0 : 1
```

It is evident that utilizing set operations on HllSets results in a slight degradation of approximation: 275 versus 269 for union and 0 versus 1 for intersection.

Proving Fundamental Set properties

Fundamental properties:

Commutative property

1. (A ∪ B) = (B ∪ A)
2. (A ∩ B) = (B ∩ A)

Associative property

3. (A ∪ B) ∪ C) = (A ∪ (B ∪ C))
4. (A ∩ B) ∩ C) = (A ∩ (B ∩ C))

Distributive property:

5. ((A ∪ B) ∩ C) = (A ∩ C) ∪ (B ∩ C)
6. ((A ∩ B) ∪ C) = (A ∪ C) ∩ (B ∪ C)

Identity:

7. (A ∪ Z) = A
8. (A ∩ U) = A

Some additional laws:

Idempotent laws:

1. (A ∪ A) = A
3. (A ∩ A) = A

```
Python
A = hll_1
B = hll_2
C = hll_3

# Defining local empty Set
Z = set.HllSet{10}()

# Defining local universal Set
U = A ∪ B ∪ C

print("\n 1. (A ∪ B) = (B ∪ A): ", count(A ∪ B) == count(B ∪ A))
print("\n 2. (A ∩ B) = (B ∩ A): ", count(A ∩ B) == count(B ∩ A))
print("\n 3. (A ∪ B) ∪ C) = (A ∪ (B ∪ C)): ", count((A ∪ B) ∪ C) == count(A
∪ (B ∪ C)))
print("\n 4. (A ∩ B) ∩ C) = (A ∩ (B ∩ C)): ", count((A ∩ B) ∩ C) == count(A ∩
(B ∩ C)))
print("\n 5. ((A ∪ B) ∩ C) = (A ∩ C) ∪ (B ∩ C): ", count(((A ∪ B) ∩ C)) ==
count((A ∩ C) ∪ (B ∩ C)))
print("\n 6. ((A ∩ B) ∪ C) = (A ∪ C) ∩ (B ∪ C): ", count(((A ∩ B) ∪ C)) ==
count((A ∪ C) ∩ (B ∪ C)))
print("\n 7. (A ∪ Z) = A: ", count(A ∪ Z) == count(A))
print("\n 8. (A ∩ U) = A: ", count(A ∩ U) == count(A))
print("\n 9. (A ∪ A) = A: ", count(A ∪ A) == count(A))
print("\n10. (A ∩ A) = A: ", count(A ∩ A) == count(A))
```

Here is the output demonstrating that all the specified set laws are satisfied:

```
Python
 1. (A ∪ B) = (B ∪ A): true
 2. (A ∩ B) = (B ∩ A): true
 3. (A ∪ B) ∪ C) = (A ∪ (B ∪ C)): true
 4. (A ∩ B) ∩ C) = (A ∩ (B ∩ C)): true
 5. ((A ∪ B) ∩ C) = (A ∩ C) ∪ (B ∩ C): true
 6. ((A ∩ B) ∪ C) = (A ∪ C) ∩ (B ∪ C): true
 7. (A ∪ Z) = A: true
 8. (A ∩ U) = A: true
 9. (A ∪ A) = A: true
10. (A ∩ A) = A: true
```

Module: entity.jl

This module supports a new structure called Entity, which encapsulates metadata using HllSets. This initiative builds on Mike Saint-Antoine's work with SimpleGrad.jl, adapted from Andrey Karpathy's MicroGrad project. Unlike the original frameworks that use numerical embeddings in their neural networks, our approach replaces them with HllSets.

Entity: formal introduction

Let us begin with a more formal definition of an Entity.

Python

```
mutable struct Entity{P}
        sha1::String
        hll::HllSets.HllSet{P}
        grad::Float64
        op::Union{Operation{FuncType, ArgTypes}, Nothing} where {FuncType,
        ArgTypes}
        # Constructor with keyword arguments
        function Entity{P}(hll::HllSets.HllSet{P}; grad=0.0, op=nothing) where
        {P}
            sha1 = string(HllSets.id(hll))
            new{P}(sha1, hll, grad, op)
        end
end
```

Any object instantiated from this struct will be an instance of Entity. When comparing this definition to the struct proposed by Mike S.-A., we observe several modifications: a few elements have been added, and one has been omitted. Specifically, the 'data' element has been substituted with 'HllSet'. In essence, we introduce only one novel element, which is 'sha1', serving as a SHA-1 based identifier for the Entity instance.

```python
mutable struct Value{opType} <: Number
    data::Float64
    grad::Float64
    op::opType
end
```

With the introduction of the Entity, we can now formally implement the concepts of **Static** and **Dynamic** structures. The Entity enables us to distinctly separate these structures by assigning them different sets of operations tailored to their specific functionalities.

Static Structure operations

HllSet embodies the core essence of the Entity and remains immutable within any given instance of an Entity context. Static operations facilitate the creation of new Entity instances through various operations, yet they do not alter the existing instances of an Entity.

In contrast, operations within the Dynamic Structure are designed to support modifications to an Entity instance. However, these modifications adhere to the overarching principle of immutability, which will be discussed further in the subsequent section.

Currently, we have identified several operations associated with the Static Structure:

1. **Negation**: This unary operation generates a new Entity instance by reversing the sign of the 'grad' field in the provided instance.
2. **Copy**: creates a new Entity instance that is a copy of the Entity argument.
3. **Union**: A binary operation that merges the HllSets of two Entity instances, resulting in a new Entity with a combined HllSet.
4. **Intersection**: Similar to the union operation, this creates a new Entity by performing an intersection on the HllSets of the provided Entity instances.
5. **XOR** (Exclusive OR): This operation also merges HllSets from two Entity instances but uses an exclusive OR approach.
6. **Complement** (Comp): This operation produces a new Entity instance containing elements from the HllSet of the first argument that are not present in the HllSet of the second argument.

The following paragraphs present the source code for all operations mentioned. Each operation is accompanied by a corresponding 'backprop' function, which propagates changes in the 'grad' field of the resulting Entity instance to the Entity instances of the input arguments. It is important to note that the unary operations, Negation and Copy, do not have associated backprop functions.

Negation:

```python
function negation(a::Entity{P}) where {P}
	return Entity{P}(HllSets.copy!(a.hll); grad=-a.grad, op=a.op)
end
```

Copy:

```python
function copy(a::Entity{P}) where {P}
	return Entity{P}(HllSets.copy!(a.hll); grad=a.grad, op=a.op)
end
```

Union:

```python
function union(a::Entity{P}, b::Entity{P}) where {P}
	hll_result = HllSets.union(a.hll, b.hll)
	op_result = Operation(union, (a, b))
	return Entity{P}(hll_result; grad=0.0, op=op_result)
end
```

We are providing the source code for the 'backprop' function specifically for the union operation. We will not be including it for other operations as they are nearly identical at present. We have chosen to keep them this way due to planned future enhancements that will likely differentiate them.

```python
function backprop!(entity::Entity{P},
	entity_op::Union{Operation{FuncType, ArgTypes}, Nothing}=entity.op)
	where {P, FuncType<:typeof(union), ArgTypes}
```

```
        if (entity.op != nothing) && (entity.op === entity_op) && (entity_op.op
        === union)
            entity_op.args[1].grad += entity.grad
            entity_op.args[2].grad += entity.grad
        else
            println("Error: Operation not supported for terminal node")
        end
end
```

Intersection:

```Python
function intersect(a::Entity{P}, b::Entity{P}) where {P}
        hll_result = HllSets.intersect(a.hll, b.hll)
        op_result = Operation(intersect, (a, b))
        return Entity{P}(hll_result; grad=0.0, op=op_result)
end
```

XOR:

```Python
function xor(a::Entity{P}, b::Entity{P}) where {P}
        hll_result = HllSets.set_xor(a.hll, b.hll)
        op_result = Operation(xor, (a, b))
        return Entity{P}(hll_result; grad=0.0, op=op_result)
end
```

Complement:

```Python
# comp - complement returns the elements that are in the set 'a' but not in the
'b'
function comp(a::Entity{P}, b::Entity{P}; opType=comp) where {P}
        hll_result = HllSets.set_comp(a.hll, b.hll)
        op_result = Operation(comp, (a, b))
        # comp_grad = HllSets.count(hll_result)
        return Entity{P}(hll_result; grad=0.0, op=op_result)
end
```

Dynamic Structure operations

This section outlines our operational implementation of the John von Neumann Self-Reproducing Automata concept. The implemented operations are designed to facilitate the creation of self-reproducing Neural Networks (NN), which will serve as the foundation for the Self-Generative System (SGS).

The primary operation responsible for the reproduction of NN elements (nodes) will be termed the **'advance'** operation. However, before we can implement this operation, it is essential to establish several supporting operations. These supportive operations will enable us to track and manage changes within the nodes of the NN and the relationships among them.

Changes in the HllSet (or simply "set") can be characterized by three distinct subsets that collectively define the overall set. More generally, these subsets can also describe the differences between two arbitrary sets (HllSets):

1. The HllSet representing elements **added** to the new HllSet compared to the old one.
2. The HllSet representing elements **retained** in the new HllSet that were inherited from the old one.
3. The HllSet representing elements that have been **deleted** in the new HllSet compared to the old one.

It is important to note that the operations discussed are based on the **complement (comp)** operation outlined in the previous section.

The operations responsible for managing changes in HllSets also oversee the grad field within the Entity. **This distinction highlights a significant difference between Static and Dynamic Structure operations.**

Added:

```
Python
function added(current::Entity{P}, previous::Entity{P}) where {P}
       length(previous.hll.counts) == length(current.hll.counts) ||
       throw(ArgumentError("HllSet{P} must have same size"))
       hll_result = HllSets.set_comp(previous.hll, current.hll)
       op_result = Operation(added, (current, previous))
       comp_grad = HllSets.count(hll_result)
       return Entity{P}(hll_result; grad=comp_grad, op=op_result)
end
```

Here is the backprop function. It effectively propagates changes through the `grad` property of the resulting Entity instance back to the Entity instances of the arguments. Similar to static operations, the backprop functions are nearly identical to those already provided. Therefore, we will omit them from the descriptions of subsequent operations.

```
Python
function backprop!(entity::Entity{P}, entity_op::Operation{FuncType, ArgTypes})
where {P, FuncType<:typeof(added), ArgTypes}

        if (entity.op != nothing) && (entity.op === entity_op) && (entity_op.op
        === added)
            entity_op.args[1].grad += entity.grad
            entity_op.args[2].grad += entity.grad
        else
            println("Error: Operation not supported for terminal node")
        end
end
```

Retained:

```
Python
function retained(current::Entity{P}, previous::Entity{P}) where {P}
        length(previous.hll.counts) == length(current.hll.counts) ||
        throw(ArgumentError("HllSet{P} must have same size"))

        hll_result = HllSets.intersect(current.hll, previous.hll)
        op_result = Operation(retained, (current, previous))
        comp_grad = HllSets.count(hll_result)
        return Entity{P}(hll_result; grad=comp_grad, op=op_result)
end
```

AS you can see this operation uses HllSet intersection to get retained HllSet.

Deleted:

```
Python
function deleted(current::Entity{P}, previous::Entity{P}) where {P}
        length(previous.hll.counts) == length(current.hll.counts) ||
        throw(ArgumentError("HllSet{P} must have same size")
        )
        hll_result = HllSets.set_comp(current.hll, previous.hll)
        op_result = Operation(deleted, (current, previous))
        comp_grad = HllSets.count(hll_result)
        return Entity{P}(hll_result; grad=comp_grad, op=op_result)
end
```

Difference:

```Python
function diff(a::Entity{P}, b::Entity{P}) where {P}
	d = deleted(a, b)
	r = retained(a, b)
	n = added(a, b)
	return d, r, n
end
```

This operation allows us to run all three previous operations in the batch. We will use it in the following operation.

Advance:

```Python
# advance - Allows us to calculate the gradient for the advance operation
# We are using 'advance' name to reflect the transformation of the set
# from the previous state to the current state
function advance(a::Entity{P}, b::Entity{P}) where {P}
	d, r, n = diff(a, b)
	hll_res = HllSets.union(n.hll, r.hll)
	op_result = Operation(adv, (d, r, n))
	# calculate the gradient for the advance operation as
	# the difference between the number of elements in the n set
	# and the number of elements in the d set
	grad_res = HllSets.count(n.hll) - HllSets.count(d.hll)  # This is the
	simplest way to calculate the gradient

	# Create updated version of the entity
	return Entity{P}(hll_res; grad=grad_res, op=op_result)
end
```

This enhanced version of the advanced function enables us to discern trends by analyzing the dynamics of changes within added, retained, and deleted subsets. This accumulated insight allows us to forecast the future state of any given Entity instance. By implementing this function across all Entity instances within the neural network, we can anticipate alterations throughout the entire system. Specifically for the SGS, this capability means that we can successfully regenerate the SGS.

```python
"""
This version of advance operation generates new unknown set from the current
set
that we are using as previous set.
Entity b has some useful information about current state of the set:
    - b.hll - current state of the set
    - b.grad - gradient value that we are going to use to calculate the
gradient for the advance operation
    - b.op - operation that we are going to use to calculate the gradient for
the advance operation.
            op has information about how we got to the current set b.
            - op.args[1] - deleted set
            - op.args[2] - retained set
            - op.args[3] - added set
We are going to use this information to construct the new set that represents
the unknown state of the set.
"""
function advance(::Colon; b::Entity{P}) where {P}
       # Create a new empty set
       hll_res = HllSets.create(HllSets.size(a.hll))
       op_result = Operation(adv, (a, hll_res))
       # calculate the gradient for the advance operation as
       # the number of elements in the a set
       grad_res = HllSets.count(a.hll)  # This is the simplest way to calculate
       the gradient

       # Create updated version of the entity
       return Entity{P}(hll_res; grad=grad_res, op=op_result)
end
```

Here is the backpropagation function specifically designed to complement the advance function:

```python
function backprop!(entity::Entity{P}, entity_op::Operation{FuncType, ArgTypes})
where {P, FuncType<:typeof(adv), ArgTypes}
       if (entity.op != nothing) && (entity.op === entity_op) && (entity_op.op
       === adv)
           if entity_op.args[1].op !== nothing
               entity_op.args[1].grad += entity.grad
           end
           if entity_op.args[2].op !== nothing
               entity_op.args[2].grad += entity.grad
```

```
            end
            if entity_op.args[3].op !== nothing
                entity_op.args[3].grad += entity.grad
            end
        else
            println("Error: Operation not supported for terminal node")
        end
end
```

Backward propagation

This is still work in progress.

Evaluating Newly Implemented Operations

In this section, we will illustrate how to utilize the implemented operations:

```Python
# Generate datasets from random strings
s1 = Set(randstring(7) for _ in 1:10)
s2 = Set(randstring(7) for _ in 1:15)
s3 = Set(randstring(7) for _ in 1:100)
s4 = Set(randstring(7) for _ in 1:20)
s5 = Set(randstring(7) for _ in 1:130)

# Add datasets to HllSets
HllSets.add!(hll1, s1)
HllSets.add!(hll2, s2)
HllSets.add!(hll3, s3)
HllSets.add!(hll4, s4)
HllSets.add!(hll5, s5)
```

```Python
entity1 = HllGrad.Entity{10}(hll1)
entity2 = HllGrad.Entity{10}(hll2)
HllGrad.isequal(entity1, entity2)

# Access the type parameter P
P_type = typeof(entity1).parameters[1]
```

```
println("The type parameter P is: ", P_type)

entity1
```

And here is output:

```Unset
Entity(sha1: 8fe6a17a8c280a6716da73b194812b0ff02e1d61;
 hll_count: 11;
 grad: 0.0;
 op: nothing);
```

```Python
c = HllGrad.union(entity1, entity2)c
println(c)
```

Output:

```Unset
Entity(sha1: 04abb4ce6da6da921a395c74dc8f585e91c9580f;
 hll_count: 27;
 grad: 0.0;
 op: Main.HllGrad.Operation{typeof(union), Tuple{Entity{10},
Entity{10}}}(union, (
Entity(sha1: 8fe6a17a8c280a6716da73b194812b0ff02e1d61;
 hll_count: 11;
 grad: 0.0;
 op: nothing);


,
Entity(sha1: aaadde0638b68333c4384820a6d505f4081a20fe;
 hll_count: 16;
 grad: 0.0;
 op: nothing);

)));
```

The result of the union operation reveals not only the resulting Entity instance but also the instances of the input arguments.

Let’s run backprop function:

```python
HllGrad.backprop!(c, c.op)
println(c.op)
c
```

And it’s output:

```
Main.HllGrad.Operation{typeof(union), Tuple{Entity{10}, Entity{10}}}(union, (
Entity(sha1: 8fe6a17a8c280a6716da73b194812b0ff02e1d61;
 hll_count: 11;
 grad: 0.0;
 op: nothing);


,
Entity(sha1: aaadde0638b68333c4384820a6d505f4081a20fe;
 hll_count: 16;
 grad: 0.0;
 op: nothing);

))
Main.HllGrad.Operation{typeof(union), Tuple{Entity{10}, Entity{10}}}(union, (
Entity(sha1: 8fe6a17a8c280a6716da73b194812b0ff02e1d61;
 hll_count: 11;
 grad: 0.0;
 op: nothing);


,
Entity(sha1: aaadde0638b68333c4384820a6d505f4081a20fe;
 hll_count: 16;
 grad: 0.0;
 op: nothing);

))
```

The result is somewhat longer than before, as it includes a complete trace of the arguments as well.

Module: tokens.jl

Module: graph.jl

In this section, we will delve into the data structures and fundamental functions integral to this module, drawing insights from the `hll_graph.ipynb` Jupyter notebook.

It's crucial to understand that the modules entity.jl and graph.jl are designed to complement each other. Both modules leverage a graph structure, employing a series of operations centered around the HolSets. Their primary distinction lies in the additional attributes that define the specific purposes of each module. Moreover, there is a direct correspondence between the structures in entity.jl and graph.jl; a HllSet identified by a specific sha1 in entity.jl will correspond to an identical HllSet with the same sha1 in graph.jl.

In the initial cell, we are configuring the environment:

```
Python
include("src//graph.jl")
using ..HllGraph
using ..HllSets
using Redis
using EasyConfig
using UUIDs
using Random

conn = Redis.RedisConnection()
```

Next, let's proceed with creating some HllSets:

```
Python
# Initialize test HllSets
hll1 = HllSets.HllSet{10}()
hll2 = HllSets.HllSet{10}()
hll3 = HllSets.HllSet{10}()
hll4 = HllSets.HllSet{10}()
hll5 = HllSets.HllSet{10}()
```

```
# Add datasets to HllSets
HllSets.add!(hll1, s1)
HllSets.add!(hll2, s2)
HllSets.add!(hll3, s3)
HllSets.add!(hll4, s4)
HllSets.add!(hll5, s5)
```

We are now ready to create some graph nodes:

```Python
node1 = HllGraph.Node(HllSets.id(hll1), "node1", HllSets.dump(hll1))
node2 = HllGraph.Node(HllSets.id(hll2), "node2", HllSets.dump(hll2))
node3 = HllGraph.Node(HllSets.id(hll3), "node3", HllSets.dump(hll3))
node4 = HllGraph.Node(HllSets.id(hll4), "node4", HllSets.dump(hll4))
node5 = HllGraph.Node(HllSets.id(hll5), "node5", HllSets.dump(hll5))

graph = HllGraph.Graph([node1, node2, node3, node4, node5], [])
```

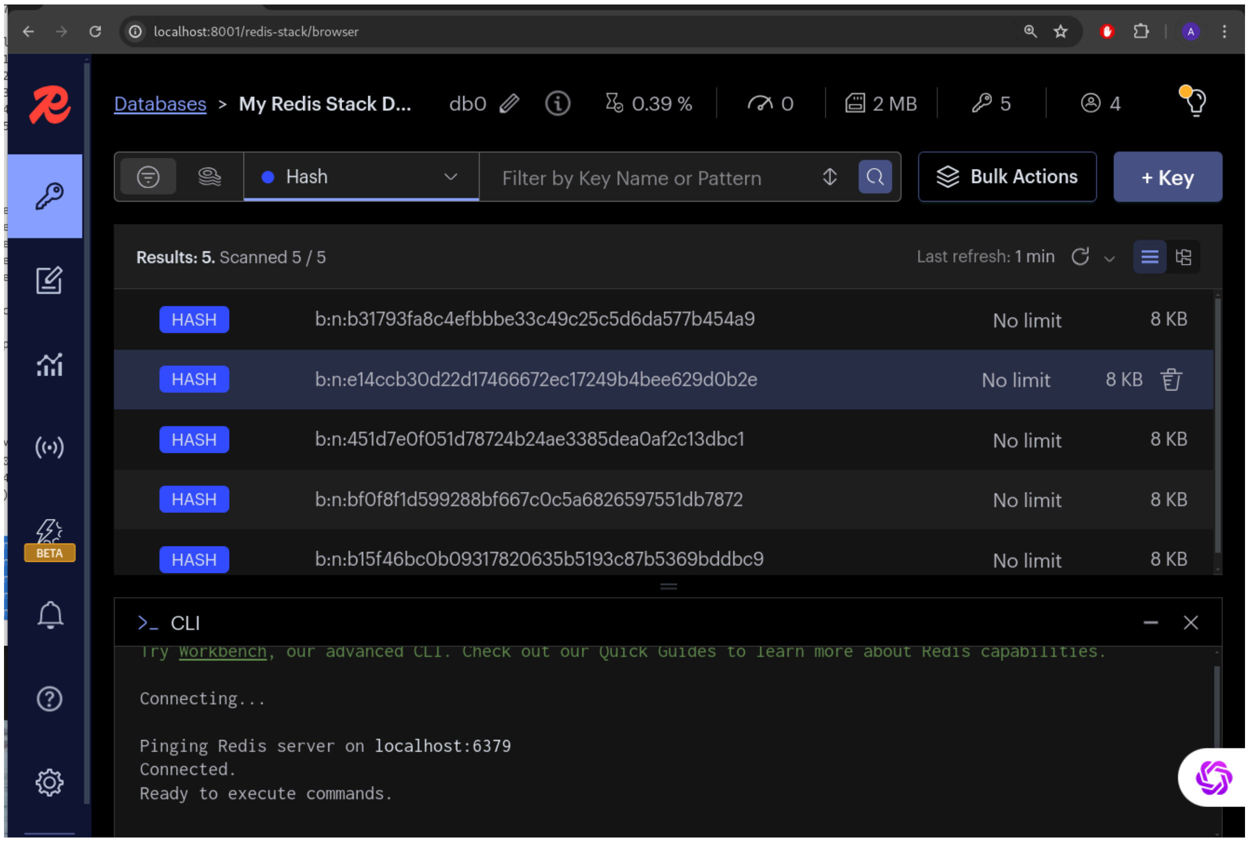
Here is the output generated by this code:

```Python
Main.HllGraph.Graph(
Main.HllGraph.Node[
Node(5c94a3ac94480f7b4c0e40294231b7ada6323388; ["node1"]; 10),
Node(abb71c6a26339830f1d58fe63c09e5caae3b3705; ["node2"]; 16),
Node(ce7cd9b5aa1aa0f26d571fa21346badb8080bb08; ["node3"]; 100),
Node(6648e755fa0d54e4122227b2ab4b81d1b768d6fb; ["node4"]; 18),
Node(2cd21d2debc73eeb9b8bffb7532cb08d176b7c95; ["node5"]; 134)],
Main.HllGraph.Edge[])
```

As you can see we have 5 nodes and no edges because we didn’t make them yet.
As depicted, we currently have 5 nodes without any edges as they have not been created yet.
Next, let us proceed to transfer the established nodes to RediSearch.

```Python
HllGraph.set_node(conn, node1, "b")
HllGraph.set_node(conn, node2, "b")
HllGraph.set_node(conn, node3, "b")
HllGraph.set_node(conn, node4, "b")
HllGraph.set_node(conn, node5, "b")
```

Here is a snapshot from RedisInsight displaying these nodes. It is important to note that they are not included in the RediSearch index as it has not been created yet.


localhost:8001/redis-stack/browser
Databases > My Redis Stack D...
db0
0.39 %
0
2 MB
5
4
Hash
Filter by Key Name or Pattern
Bulk Actions
+ Key
Last refresh: 1 min
BETA
CLI

Results: 5. Scanned 5 / 5

HASH	b:n:b31793fa8c4efbbbe33c49c25c5d6da577b454a9	No limit	8 KB
HASH	b:n:e14ccb30d22d17466672ec17249b4bee629d0b2e	No limit	8 KB
HASH	b:n:451d7e0f051d78724b24ae3385dea0af2c13dbc1	No limit	8 KB
HASH	b:n:bf0f8f1d599288bf667c0c5a6826597551db7872	No limit	8 KB
HASH	b:n:b15f46bc0b09317820635b5193c87b5369bddbc9	No limit	8 KB

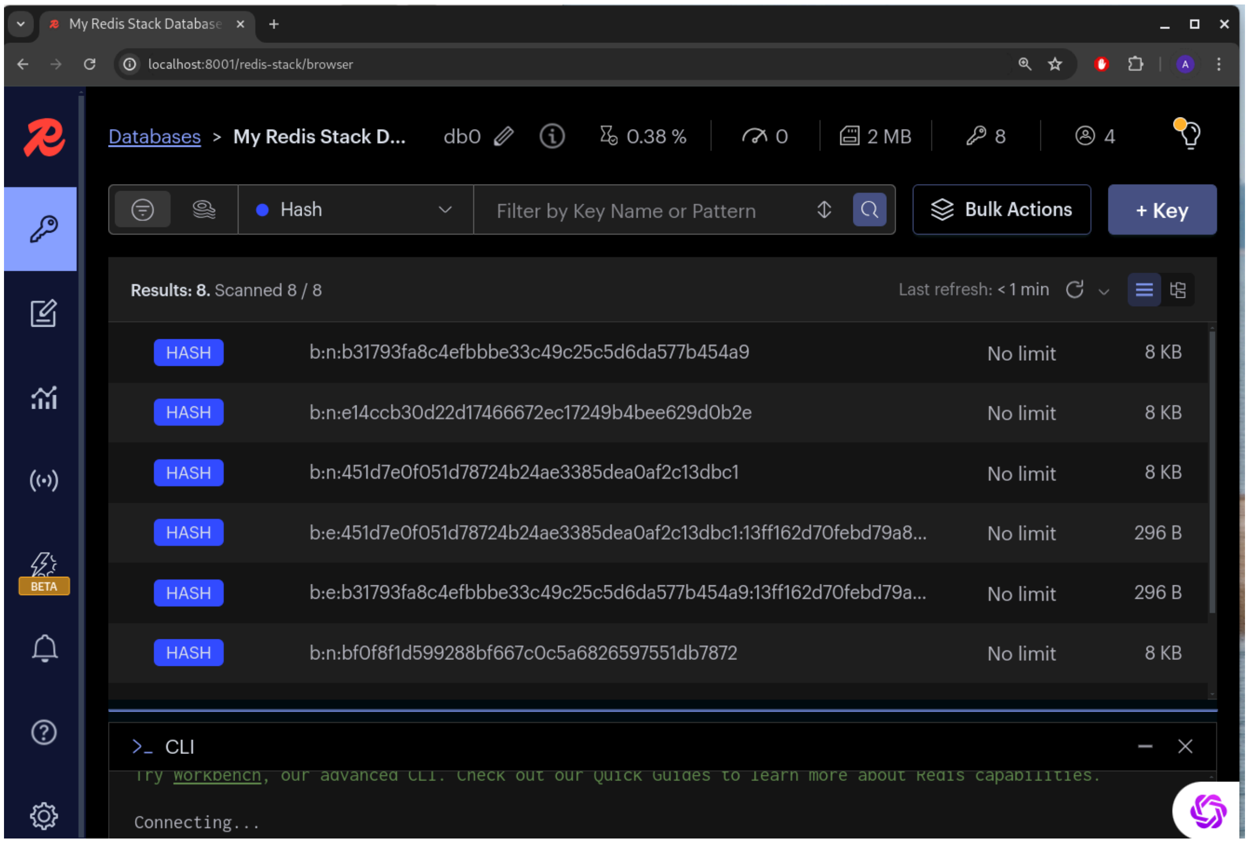
```
Try Workbench, our advanced CLI. Check out our Quick Guides to learn more about Redis capabilities.

Connecting...

Pinging Redis server on localhost:6379
Connected.
Ready to execute commands.
```

HllGraph fully supports all HllSet set operations. Let's demonstrate by creating a Union of two nodes, node1 and node2. Below is the code to achieve this:

```Python
union_node = HllGraph.union_nodes(conn, node1, node2, "union", "b")
```

HllGraph offers two variations for each set_node and set_edge operation: one that generates only the resulting node and necessary edges linking it to the node-arguments, and another that not only performs the first task but also generates a hash in Redis. The latter version of set functions includes an extra parameter for establishing a connection to Redis (referred to as ‘conn’ in union_nodes(conn, node1, node2, . . .).

Upon completion of this operation, there will be two edges in the list of hashes (refer to hashes with the prefix ‘b:e:. . . . .’):


My Redis Stack Database
localhost:8001/redis-stack/browser
Databases > My Redis Stack D... db0
0.38 %
0
2 MB
8
4
Hash
Filter by Key Name or Pattern
Bulk Actions
+ Key
Last refresh: < 1 min
BETA
CLI

```
Results: 8. Scanned 8 / 8

HASH  b:n:b31793fa8c4efbbbe33c49c25c5d6da577b454a9                      No limit  8 KB
HASH  b:n:e14ccb30d22d17466672ec17249b4bee629d0b2e                      No limit  8 KB
HASH  b:n:451d7e0f051d78724b24ae3385dea0af2c13dbc1                      No limit  8 KB
HASH  b:e:451d7e0f051d78724b24ae3385dea0af2c13dbc1:13ff162d70febd79a8...  No limit  296 B
HASH  b:e:b31793fa8c4efbbbe33c49c25c5d6da577b454a9:13ff162d70febd79a...   No limit  296 B
HASH  b:n:bf0f8f1d599288bf667c0c5a6826597551db7872                      No limit  8 KB
```

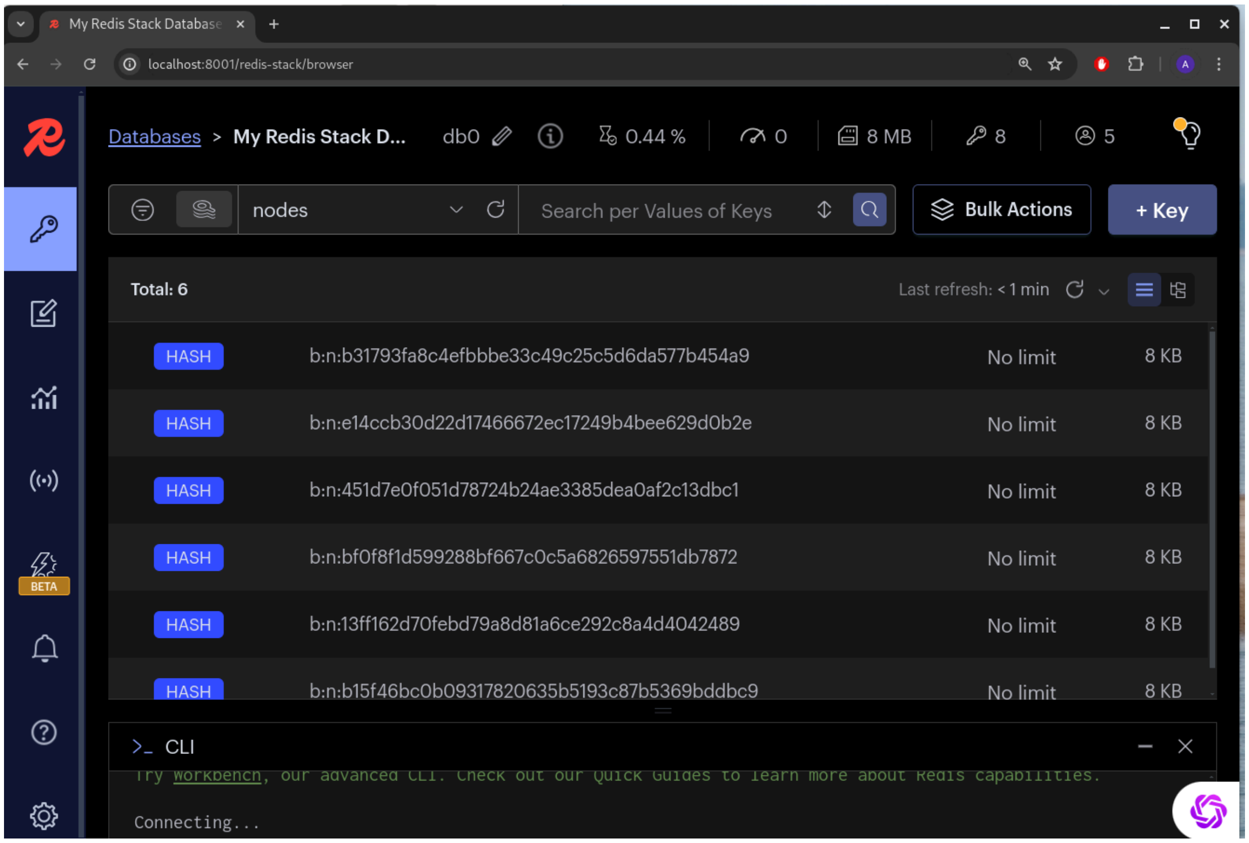
```
Try Workbench, our advanced CLI. Check out our Quick Guides to learn more about Redis capabilities.

Connecting...
```

Now, let's proceed with creating a RediSearch index to store the created nodes. To begin, we will define the schema for this index:

```Python
try
    Redis.execute_command(conn, ["FT.CREATE", "nodes", "ON", "HASH",
         "PREFIX", 1, "b:n:",     # Here is the prefix that we used in node
hashes
        "SCHEMA", "sha1", "TEXT", "labels", "TEXT", "searchable", # We skipped
dataset field
        "VECTOR",
        "HNSW",
        "16",
        "TYPE",
        "FLOAT32",  # Notice we are using Float32 instead of UInt64
```

```
            "DIM",
            "1024",     # The size search vector the same as HllSet
            "DISTANCE_METRIC",
            "COSINE",
            "INITIAL_CAP",
            "50000",
            "M",
            "40",
            "EF_CONSTRUCTION",
            "100",
            "EF_RUNTIME",
            "20",
            "EPSILON",
            "0.8"])
catch e
    println(e)
end
```

We have added comments to certain fields in the schema to provide clarification. One change we made was switching from UInt64 to FLOAT32 in HllSet. This adjustment was necessary to meet the requirements of RediSearch. Although we initially had a precision of 64 bits, we decided to downgrade to 32 bits in order to conserve memory. Unfortunately, RediSearch does not support 16 bit floats, so further reduction was not possible. Below is the index for the b:nodes:


My Redis Stack Database
localhost:8001/redis-stack/browser
Databases > My Redis Stack D... db0
0.44 %
0
8 MB
8
5
nodes
Search per Values of Keys
Bulk Actions
+ Key
BETA

Total: 6 Last refresh: < 1 min

HASH	b:n:b31793fa8c4efbbbe33c49c25c5d6da577b454a9	No limit	8 KB
HASH	b:n:e14ccb30d22d17466672ec17249b4bee629d0b2e	No limit	8 KB
HASH	b:n:451d7e0f051d78724b24ae3385dea0af2c13dbc1	No limit	8 KB
HASH	b:n:bf0f8f1d599288bf667c0c5a6826597551db7872	No limit	8 KB
HASH	b:n:13ff162d70febd79a8d81a6ce292c8a4d4042489	No limit	8 KB
HASH	b:n:b15f46bc0b09317820635b5193c87b5369bddbc9	No limit	8 KB

```
>_ CLI
Try Workbench, our advanced CLI. Check out our Quick Guides to learn more about Redis capabilities.

Connecting...
```

Executing a basic search query will retrieve all nodes:

```
Python
ft.search nodes * return 2 sha1 labels
1) "6"
2) "b:n:b31793fa8c4efbbbe33c49c25c5d6da577b454a9"
3) 1) "sha1"
   2) "b31793fa8c4efbbbe33c49c25c5d6da577b454a9"
   3) "labels"
   4) "[\"node1\"]"
4) "b:n:e14ccb30d22d17466672ec17249b4bee629d0b2e"
5) 1) "sha1"
   2) "e14ccb30d22d17466672ec17249b4bee629d0b2e"
   3) "labels"
   4) "[\"node3\"]"
6) "b:n:451d7e0f051d78724b24ae3385dea0af2c13dbc1"
7) 1) "sha1"
   2) "451d7e0f051d78724b24ae3385dea0af2c13dbc1"
   3) "labels"
   4) "[\"node2\"]"
8) "b:n:bf0f8f1d599288bf667c0c5a6826597551db7872"
```

```
9) 1) "sha1"
   2) "bf0f8f1d599288bf667c0c5a6826597551db7872"
   3) "labels"
   4) "[\"node5\"]"
10) "b:n:13ff162d70febd79a8d81a6ce292c8a4d4042489"
11) 1) "sha1"
   2) "13ff162d70febd79a8d81a6ce292c8a4d4042489"
   3) "labels"
   4) "[\"union\"]"
12) "b:n:b15f46bc0b09317820635b5193c87b5369bddbc9"
13) 1) "sha1"
   2) "b15f46bc0b09317820635b5193c87b5369bddbc9"
   3) "labels"
   4) "[\"node4\"]"
```

Still unsettled things

Balancing memory footprint and calculations

The use of Relational Algebra within HllSet offers a vast array of possibilities for creating new HllSets, thereby generating new Graph nodes. The Graph maintains the relationships between node arguments and the resulting node, allowing for the regeneration of the dataset for the resulting node based on the datasets of the argument nodes. One potential solution could be to store the resulting dataset in memory when utilizing the graph and omit it when saving to disk.

Utilizing Multiple RediSearch Indexes for Nodes, Edges, and Tokens

Maintaining separate indices for each SGS data type—Node, Edge, and Token—presents challenges in querying data. Moreover, RediSearch prohibits using the same hash across multiple indices.

RediSearch links each index to a unique prefix or pattern within the Redis keys. When a hash key aligns with an index's designated pattern, RediSearch automatically incorporates that hash into the index based on its predefined schema. As each hash key is tied to a single full key name, it can correspond to only one index pattern at a time.

Nevertheless, there are strategies to navigate this limitation and enable a hash to be searchable across multiple indices:

1. **Duplicate Data**: Replicating the hash under different keys corresponding to various indices' patterns increases storage demands due to data redundancy.
2. **Alias or Reference Keys**: Establish lightweight keys that reference the original hash and adhere to the indices' patterns. This method introduces additional complexity in managing references and can complicate the processes of data retrieval and updates.
3. **Schema Design**: Carefully plan your index schemas and key naming strategies to accommodate your query needs with minimal indices. This may involve utilizing more sophisticated queries within a single index rather than distributing data across multiple indices.

Alternatively, the `**RENAME**` command allows for reassigning a hash to a different index by modifying its key to fit an alternate pattern. This operation, which does not replicate the data, is particularly effective in scenarios where data transitions through various states or stages and requires searchability in different contexts. **However, it does not support the simultaneous presence of the same hash in more than one index.**

Technical References

1. https://stackoverflow.com/questions/19729831/angle-between-3-points-in-3d-space
2. https://github.com/yahoo/redislite
3. http://bitop.luajit.org/index.html
4. https://github.com/CBaquero/random/blob/master/README.md

References

1. https://mikesaint-antoine.github.io/SimpleGrad.jl/under_the_hood/
2. https://github.com/karpathy/micrograd
3. https://www.youtube.com/watch?v=VMj-3S1tku0
4. https://nnfs.io/
5. https://www.cnbc.com/2022/09/21/why-elon-musk-says-patents-are-for-the-weak.html
6. https://github.com/microsoft/autogen
7. https://redis.io/docs/latest/commands/ttl/
8. https://medium.com/javarevisited/how-redis-ttl-works-c77785d49a96
9. https://redis.io/blog/cache-eviction-strategies/
10. https://www.linkedin.com/pulse/caching-strategies-eviction-policies-redis-asim-hafeez/
11. https://redis.io/wp-content/uploads/2021/12/caching-at-scale-with-redis-updated-2021-12-04.pdf
12. https://medium.com/intuition/analog-computation-of-linear-algebra-theory-and-implementation-b52209c49c1c
13. https://levelup.gitconnected.com/skeleton-recall-loss-is-the-new-breakthrough-in-segmentation-b1ce43c093f4

14. https://arxiv.org/pdf/2404.03010 (SkeletonRecall)
15. https://github.com/MIC-DKFZ/Skeleton-Recall/blob/master/nnunetv2/inference/examples.py
16. https://pypi.org/project/NiftyNet/
17. https://arxiv.org/pdf/1707.03237v3 (NiftyNet paper)
18. https://arxiv.org/pdf/2003.07311 (clDice - a Novel Topology-Preserving Loss Function)
19. https://arxiv.org/pdf/2309.02527 (skeletonization algorithm for gradient-based optimization)
20. https://www.nature.com/articles/s41592-020-01008-z.epdf?sharing_token=IRTK46PbU4jVyrO5IHF7gNRgN0jAjWeI9jnR3ZoTv0MPk71Wg6vREIdiNjHEbU89De36tbNDxGQNyPhRVISxhGpdyXfQ2Y7Gni-kjqYmrX9f02ybBDE5znbj4_1vdV2iNyZixT9ry1IwwGNYHNthDKBUH2nlsB4T5UvoiBK3bzXb5yGOF_hxEMGJR_X0ezJEJxwd2a9D4onfzKJAqGMAowX8OhqhGgI5wBalJei_E4JoGU2MCbLIN0z53uZ8YNsyv8cxwb1YPA3uQrLybR7YxvbpANm5srQ7FoXhVtjmB-9Viw6uPpcHcJgIZ0sm1Zs92nKTMfqDFjhKoSGykSP1LA%3D%3D&tracking_referrer=levelup.gitconnected.com
21. https://arxiv.org/pdf/1409.0575 (ImageNet Large Scale Visual Recognition Challenge)
22. https://en.wikipedia.org/wiki/Convolutional_neural_network#Notable_libraries
23. https://en.wikipedia.org/wiki/Lateral_geniculate_nucleus

Appendix 1

I would like to discuss the core aspects of SGS development strategy regarding Intellectual Properties, drawing insights from notable industry practices [5].

During a tour, when Jay Leno inquired whether SpaceX had patented the material used in constructing its spacecraft, Elon Musk clarified that his company does not focus on patenting. "We don't really patent things," Musk stated. He expressed a general disinterest in patents, remarking, "I don't care about patents. Patents are for the weak."

Musk views patents primarily as obstacles to innovation, suggesting that they are often employed to hinder competition rather than to foster advancement. "Patents are generally used as a blocking technique," he explained, likening them to "landmines in warfare." According to Musk, patents do not promote progress; instead, they merely prevent others from pursuing similar paths.

This perspective is consistent with Musk's previous statements. In a 2014 memo to Tesla employees, he attributed the company's potential for success to its ability to attract and motivate top engineering talent, rather than to its portfolio of patents. He criticized the patent system for hampering technological advancement, maintaining the status quo for large corporations, and benefiting the legal field rather than the inventors themselves.

Furthermore, Tesla's legal page contains a commitment that underscores this philosophy: the company promises not to engage in patent litigation against anyone who wishes to use its technology in good faith.

These insights into Musk's approach to intellectual property highlight a strategic focus on innovation and open access, rather than on securing competitive edges through legal protections.

Appendix 2

This excerpt from upcoming book "Multi-Agent Systems with AutoGen. Victor Dibia with Chi Wang"
(https://www.manning.com/books/multi-agent-systems-with-autogen?trk_msg=F7K7PGD9NTNK96NH8TAG1G6MG0&trk_contact=5CHBHO5KD3QC9542IAR8GD0K70&trk_sid=QAUM56FJ70F98U7R3RQP0CK6IO&trk_link=OOK1PRA1NGC4T3HA0B9C92RSE4&utm_source=Listrak&utm_medium=Email&utm_term=https%3a%2f%2fwww.manning.com%2fbooks%2fmulti-agent-systems-with-autogen&utm_campaign=The+Weekly+Roundup+July+20%e2%80%9426)

AN ADAPTIVE MULTI-AGENT APPLICATION EXAMPLE

Consider the following scenario: You are a developer, and your task is to create an application that can book a flight to a specific destination at the best price. However, there is a constraint - the selected destination can only be booked through a specific airline called specialairlines.com, which doesn't have an API. Instead, they have two primary interfaces designed for users to book flights - a web interface and a mobile app. Beyond the apparent requirements for the task - conducting a search for the best price, booking the flight, and ensuring the booking is successful - there are several complexities that need to be addressed.

Specifically, a successful agent must be able to navigate an end-user interface. While this may seem trivial, it is relatively complex, as the agent must understand the content of the interface (e.g., by processing the HTML elements or pixels in a screenshot of the page), derive an action to take (e.g., click a button, fill a form), and then verify that the action was successful (e.g., by checking for a success message). Note that every action taken alters the state of the interface, which, in turn, affects the next set of actions that can be taken.

The agent must repeatedly take action and be able to adapt to other changes in the interface (e.g., if the button location changes or the form fields are updated) and be able to recover from errors (e.g., if the booking fails, the agent must be able to retry the booking) until the task is completed. Finally, the agent must communicate effectively with other agents (e.g., a payment agent that processes the payment) and humans (e.g., if the agent is unable to complete the booking, it may need to ask the user for help).

The scenario above illustrates a situation where the solution to the task is not known a priori and must be discovered through a series of actions, each of which affects the state of the task. This is a classic example of a complex task that requires reasoning, acting, adapting, and communicating across multiple agents to solve, a task that cannot be addressed by current LLM-enabled applications and is a potential candidate for a multi-agent system.