HllSet commit and Self Reproductive System

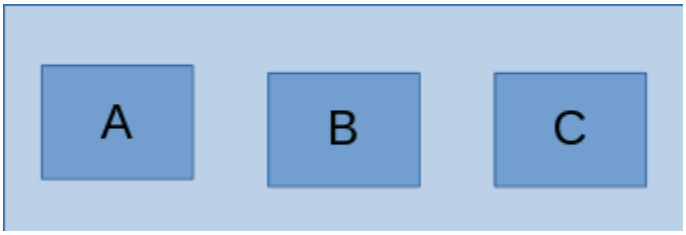
> *Things are great and small, not only by the will of fate and circumstances, but also according to the concepts they are built on. —Kozma Prutkov [2]*

This article begins by exploring the concept of self-reproducing automata, as introduced by John von Neumann (refer to [1]). According to the Wikipedia entry [2], the initial studies on self-reproducing automata date back to 1940. Over the span of more than 80 years, significant advancements have been made in this field of research and development, bringing us closer to the realization of systems capable of self-reproduction, which we will refer to as **self-generative systems** (**SGS**).

The purpose of this article is to demonstrate a proof of concept by developing a Metadatum SGS. Metadatum is a metadata management system that leverages Julia and Neo4J Graph DB as its operational environment.

0. Introduction to John von Neumann theory of self-reproduction

John von Neumann's concept of self-replicating systems is intriguingly straightforward. Imagine a system composed of three distinct modules: A, B, and C.



Module A acts as a Universal Constructor, capable of crafting any entity based on a provided blueprint or schema.

Module B functions as a Universal Copier, able to replicate any entity's detailed blueprint or duplicate an entity instance.

Module C, the Universal Controller, initiates an endless cycle of self-replication by activating modules A and B.

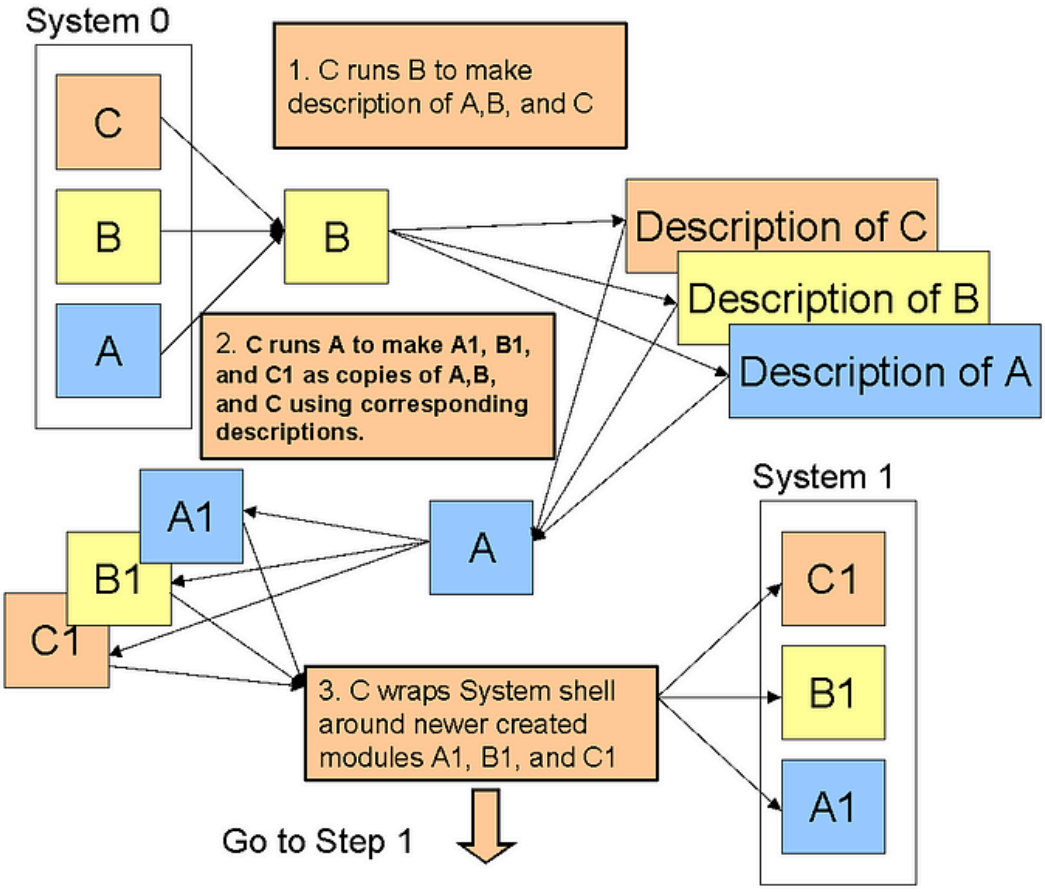


Figure 2.

The self-replication process begins with Module C, leading to the creation of an identical system, System 1, from the original System 0. This new system is equally capable of initiating its own self-replication cycle, adhering to the same algorithm.

From this analysis, several key insights emerge.

Firstly, the self-replication algorithm is sufficiently generic to be implemented across various platforms.

Secondly, Module C's infinite loop can orchestrate the self-replication process.

Lastly, this algorithm represents a theoretical framework for system upgrade automation, or self-upgrading.

However, in its basic form, a self-replicating system merely clones itself. To enhance its utility, a fourth module, Module D, is introduced. This module enables interaction with the system's environment and access to its resources, effectively functioning as an application within an operating system composed of Modules A, B, and C.

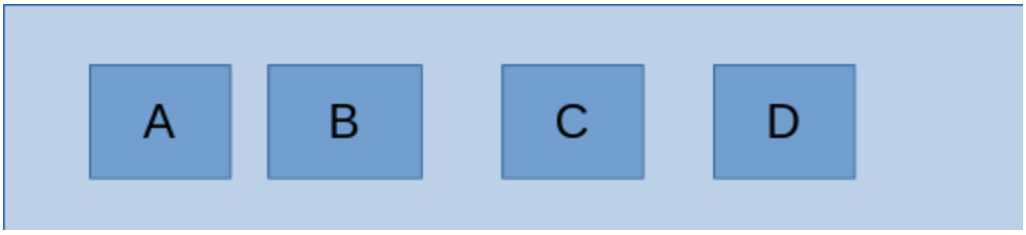


Figure 3.

Additionally, a special unit for storing module descriptions, termed the System Description, is incorporated. This upgraded self-replication process, depicted in subsequent figures, involves creating copies of each module description (A, B, C, D) alongside the System Description unit. This leads to the creation of an upgraded system version, which then replaces the old version, thus achieving a new iteration of the system.

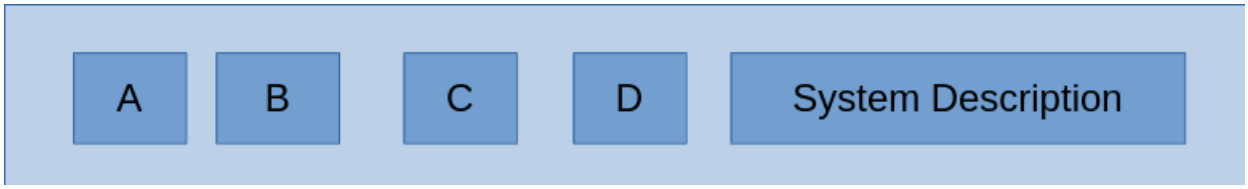


Figure 4.

This enhanced model differs from John von Neumann's original concept by introducing a dedicated unit for system descriptions, allowing the system to interact with its environment via Module D, and modifying the role of Module B to work solely with the System's Description.

Despite these advancements, the initial creation of the first self-replicating system remains an unsolved "Chicken and Egg" dilemma. Yet, as we draw parallels between this abstract model and software systems, we see opportunities for applying self-replication in managing software application life cycles.

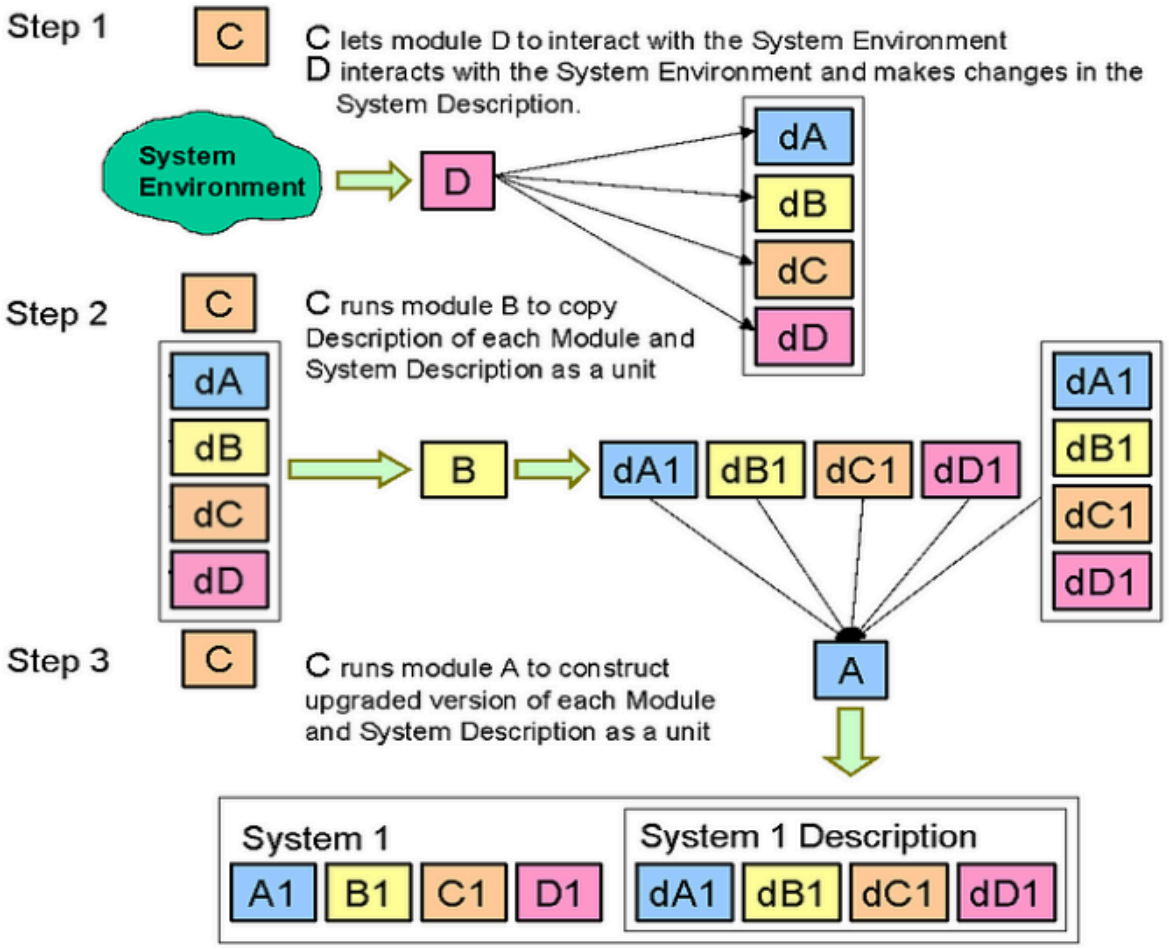


Figure 5.

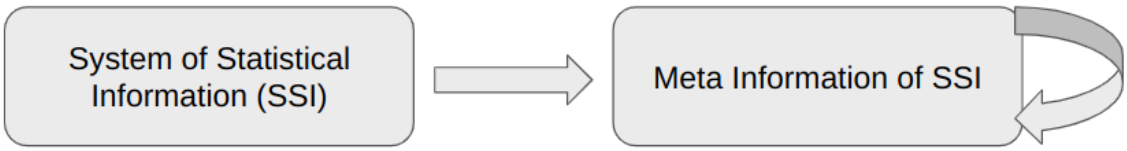
In software terms, Modules A, B, and C could represent engines facilitating continuous service processes, such as database engines, servers, or runtime environments. Module A could serve as a compiler or interpreter, generating processes based on source code. Module B might support reflection, serialization, and data buffering, ensuring system persistence and enabling development, evolution, and backup. Module D would represent application software, facilitating user and environment interaction.

Ultimately, viewing self-generative systems (SGS) as a means to standardize and automate the development cycle of software applications—from development to testing, and testing to production—opens up exciting possibilities for autonomous software development.

The purpose of this document is to utilize the concept of SGS within the Metadata Management System to analyze the Socio-Economic System.

1. Introduction to Metadata

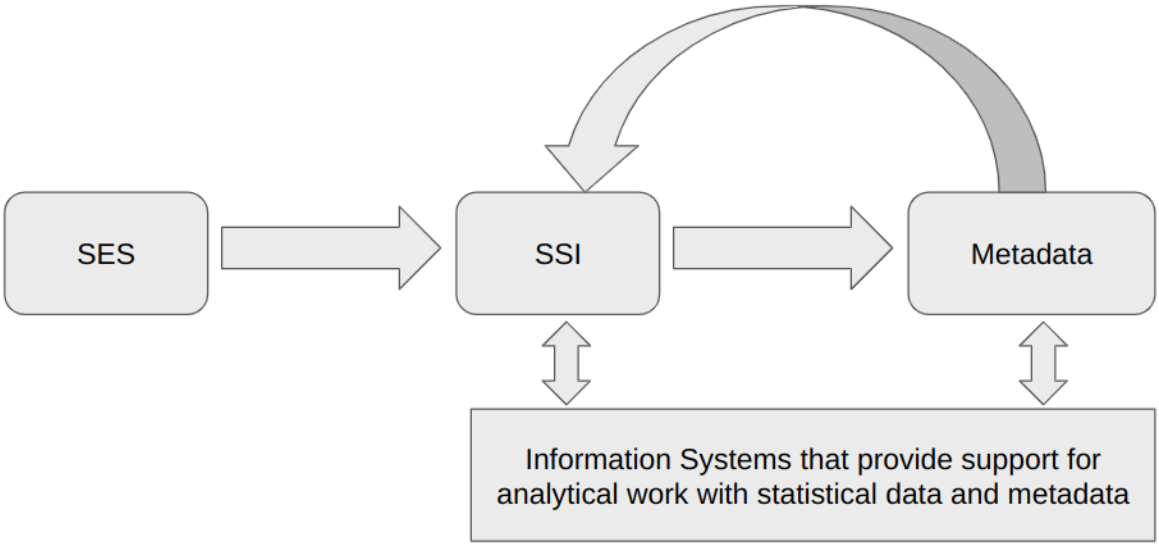
Metadata is essentially information about information. It encompasses any type of digital content that can be stored on a computer, including documents, databases, images, videos, audio files, and sensor signals. **From a metadata standpoint, all of these forms of data are treated equally and hold the same significance**.



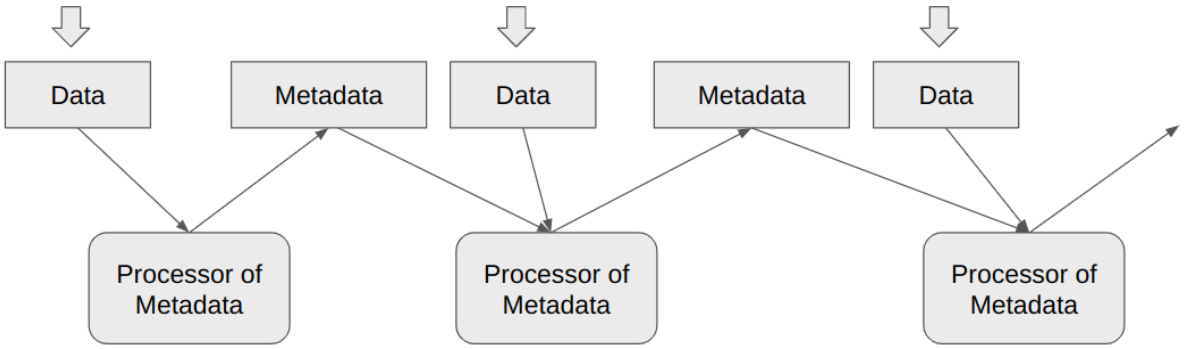
What, then, can be said about **the concept of metadata for metadata**? Essentially, metadata refers to data about data. **Thus, when we discuss metadata derived from metadata, we are essentially discussing the same entity**.

This point is crucial. **We propose to manage both the metadata of original data and the metadata of metadata through a singular metadata system**. This approach is visually represented in the figure, where we depict the metadata loop closing in on itself.

Furthermore, we delineate the ongoing process of generating metadata, which evolves over time both from the initial data and from metadata previously created. This cyclical process highlights the dynamic and iterative nature of metadata generation.



By integrating socio-economic systems (SES) mapping into statistical information systems (SIS) through statistical observation, and then mapping SIS into Metadata, we can develop a comprehensive and generalized scheme.



The diagram illustrates the SES → SIS and SIS → (Metadata), clearly demonstrating how it maintains the integrity and structural relationships within the displayed system, as discussed earlier.

It is also the rationale behind our proposal to use SGS as the cornerstone of the Metadata Management System.

This approach not only highlights the crucial characteristic of statistics as a system— its closed nature in relation to the Statistical Information System (SIS)—but also ensures that **any data encompassed by or generated within the system throughout its processing phase is seamlessly integrated into the SIS. This integration process is precise, safeguarding the original data's structure.**

In the contemporary landscape, statistical information transcends traditional statistical indicators and tables. It leverages an array of computer technologies for presenting statistical data and the outcomes of statistical analyses. This encompasses everything from basic tables and charts to sophisticated multimedia and virtual reality presentations.

When it comes to metadata, it encompasses all forms of data. For these datasets, it's imperative to create descriptive metadata elements and to forge connections among these elements.

Conceptually, metadata can be visualized as a graph. Within this graph, metadata elements are depicted as nodes, while the links between these elements are represented by the graph's edges. This structure facilitates a comprehensive and interconnected representation of metadata, enhancing the understanding and utilization of statistical information.

But before moving to the graph let’s look at the very short introduction to the HllSets (HyperLogLog Sets). For more information please look at my previous post [3].

2. HllSets

In a previous post [4], I introduced HllSets, a data structure based on the HyperLogLog algorithm developed by Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier [6]. In that post, we demonstrated that HllSets adhere to all the fundamental properties of Set theory.

The fundamental properties that HllSets complies with are as follows:

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 ∪ ∅) = A
8. (A ∩ U) = A

In addition to these fundamental properties, HllSets also satisfy the following additional laws:

Idempotent Laws:
9. (A ∪ A) = A
10. (A ∩ U) = A

To see the source code that proves HllSets satisfies all of these requirements, refer to **lisa.ipynb**[8].

3. Mapping HllSets into Graph DB

The graph can be defined as follows:

$$\mathbf{G = \{V, E\}},$$

Where

G - graph;
V - is the set of graph nodes, which in our case will represent HllSet;
E - is the set of edges connecting connected nodes of the graph.

Node v ∈ **V,** can be described as **struct** in programming languages C++, Rust, Julia or **Dict** in Python language. In the code snippet below, we have used Julia notation.

```
Python
struct Node <: AbstractGraphType
        sha1::String             # SHA1 hash ID, calculated using subset of
metadata
        labels::Vector{String}   # array of labels
        d_sha1::String           # SHA1 hash calculated using the HllSet
        card::Int                # cardinality of HllSet
        dataset::Vector{Int}     # dump of HllSet (compact presentation of
HllSet)
        props::Config            # array of additional properties presented as
JSON
 end
```

The edges of the graph describe the connections between nodes. Any pair of nodes can have more than one edge, and each edge has a direction from the source to the target.

```
Python
struct Edge <: AbstractGraphType
        source::String # sha1 of the source node
        target::String # sha1 of the target node
        r_type::String # label of the edge
        props::Config  # Additional properties presented as JSON
end
```

4. Life cycle, Transactions, and Commits

> *If everything past were present, and the present continued to exist with the future, who would be able to make out: where are the causes and where are the consequences?—Kozma Prutkov [2]*

This section will delve into some key technical details that are crucial for developing SGS as a programming system.

4.1. Transactions

In this section, we employ the "transaction" index (or a transactional table - t_table, if we're discussing databases) as an alternative to the System Description found in the self-reproduction

diagram of Chapter 0 (refer to Figure 5). The following is a flowchart that outlines the process of handling external data in the Metadatum SGS.

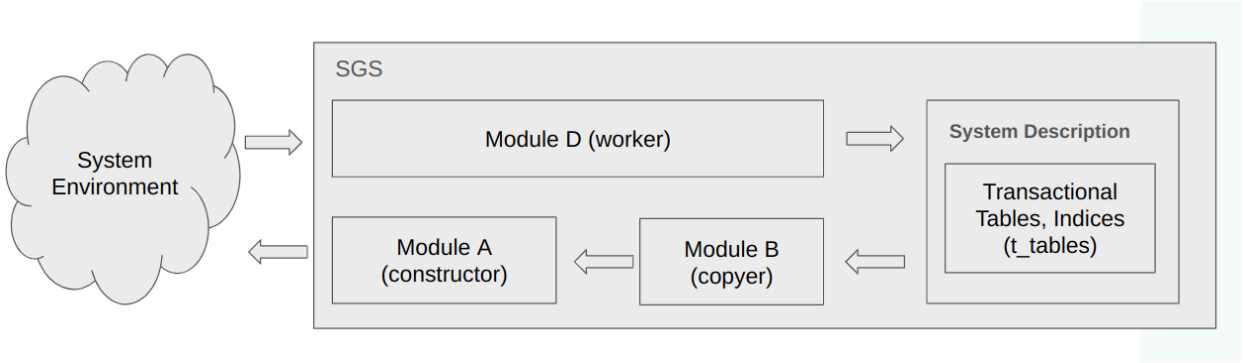


Figure 5.

Module D obtains inputs from the System Environment and records these inputs by generating records in the "transaction" index. Simultaneously, Module A, with assistance from Module B (the copier), retrieves these references from the "transaction" index. It then processes these references by engaging the appropriate processors and subsequently uploads the processed data back into the System.

It is crucial to note that SGS never directly sends incoming data to the System. Instead, it first segregates all incoming data logically into a staging area using the references in the "transaction" index.

This approach helps us achieve several objectives:

1. Clear separation between data already present in the System and new data.
2. Complete control over the processing of new data, enabling us to track completed tasks and pending work. It also facilitates support for parallel processing and recovery from failures.
3. Ability to isolate unsupported data types.
4. Strict adherence to the self-reproduction flow outlined in Chapter 0.

4.2. Commits

In the SGS (Self Generative System), each entity instance is categorized under one of three primary commit statuses, which are crucial for tracking modifications. These statuses are as follows:

1. **Head**: This status signifies that the entity instance represents the most recent modification.
2. **Tail**: An instance with this status is identified as a prior modification, indicating that it is not the latest version.
3. **Deleted**: This status is assigned to instances that have been marked as removed from the system.

To better understand how commit statuses function, consider the following illustration. The diagrams visualize the timeline of modifications, starting from the most recent (current time) at the top and progressing downwards to the earliest at the bottom.

Current time.

	Commit ID	Time	item_1	item_2	item_3	item_4	item_5	item_6
1	bcdea	3					head	head
2	cdeab	2		head	head			
3	abcde	1	head	tail		head		

Time of the previous commit. Take note of how the updated version of item_2 changed the commit state in row 2 from its original state.

	Commit ID	Time	item_1	item_2	item_3	item_4		
1	cdeab	2		head	head			
2	abcde	1	head	tail		head		

Time of the initial commit.

	Commit ID	Time	item_1	item_2		item_4		
1	abcde	1	head	head		head		

Essentially, each commit in the system carries its unique "commit forest," depicted through a distinct matrix. For every commit, there's a designated matrix. However, there's no cause for concern—these matrices are virtual. They don't need to exist physically as we can generate them as needed.

At time = 1, we observed three items: item_1, item_2, and item_4, all of which were tagged with the 'head' status, indicating their current and active state.

By time = 2, changes were made to item_2. Consequently, at this juncture, a new version of item_2 emerged within the SGS, introducing a fresh element. This new version was also tagged with the 'head' status, while the previous version's status was switched to 'tail,' indicating it's now a historical entry.

Updating is a methodical process that entails several steps:

Creating a new version of the item to be updated;

Applying the required modifications to this new version;

Saving these changes;

Establishing a connection between the new and the former version of the item.

By time = 3, two additional elements—item_5 and item_6—were introduced and similarly tagged with the 'head' status.

This mechanism of commits in the SGS crafts a narrative of system evolution. Each cycle of self-reproduction within the SGS adds new chapters to this "history book," linking back to system snapshots at the time of each commit.

In this "history book," we distinguish between 'head' and 'tail.' The 'head' represents the immediate memory and the current state of the SGS, while the 'tail' serves as an archive. Although still accessible, retrieving information from the 'tail' requires additional steps.

The commit history functions as the intrinsic timeline of Self-Generative Systems, akin to biological time in living organisms.

4.3. Static and Dynamic Metadata Structure

Data serves as a mirror of the real world, while metadata, as an abstraction of the data, serves as a reflection of the real world.

These observations rest on the underlying belief that there is a direct correspondence between the complexities of the real world and the elements within our datasets. Here, each piece of data is essentially a combination of a value and its associated attributes. When we define associative relationships based on data and its metadata, drawing on similarities between data elements, we're dealing with what's known as a **Static Data Structure**. The term "**static**" implies that these relationships are fixed; they remain constant and can be replicated as long as the data elements are described using the same attributes. Modern databases excel at mapping out these types of relationships.

Nonetheless, ***the primary aim of data analysis is to unearth hidden connections among data elements, thereby uncovering analogous relationships in the real world.*** This endeavor presupposes that the relationships we discover within our data somehow mirror those in the

real world. However, these connections are not immediately apparent—they are hidden and transient, emerging under specific conditions. As circumstances evolve, so too do the relationships among real-world elements, necessitating updates in our Data Structure to accurately reflect these changes. This leads us to the concept of a **Dynamic Data Structure**.

A **Dynamic Data Structure** emerges from the process of data analysis, facilitated by various analytical models, including Machine Learning or Artificial Intelligence. Broadly speaking, an analytical model comprises an algorithm, source data, and the resulting data. The relationships forged by these models are ephemeral and might not have real-world counterparts. Often, they represent an analyst's subjective interpretation of the real world's intricacies. These model-generated relationships constitute a Dynamic Data Structure.

The nature of a Dynamic Data Structure is inherently fluid—relationships deemed accurate yesterday may no longer hold today. Different models will vary in their relevance, and the analyst's primary challenge is to select the models that best fit the current real-world scenario and the specific aspects under consideration.

5. SGS Demo (PoC)

In this demonstration, we will leverage data and methodologies previously explored in the Enron Email Analysis Demo project [9]. For an in-depth understanding of this application, please refer to the mentioned source. Additionally, the source code is available in reference [10].

It's important to note that the conceptual design often diverges from what is implemented in the production environment, and our project is no exception. Our Self Generative System (SGS), inspired by John von Neumann's Self Reproducing model, encapsulates its core functionality within a singular commit function as shown below:

```
Python
function commit(db::Graph.DB, hdf5_filename::String, committer_name::String,
committer_email::String, message::String, props::Config)
       # Obtaining time based UUID for commit_id
       commit_id = string(uuid4())
       # Creating a recod in 'commits' table
       props = JSON3.write(props)
       commit = Graph.Commit(commit_id, committer_name, committer_email,
message, props)
       Graph.replace!(db, commit)
       # Running commit for nodes and edges
       commit_node(db, hdf5_filename, commit_id)
       commit_edge(db, hdf5_filename, commit_id)
end
```

Before we proceed, I'd like to offer a few remarks about the provided code, serving as a form of disclaimer.

The code samples may appear to be somewhat verbose, and this is a deliberate decision. At this early stage of our expansive project, we have a clear vision of our end goal, yet the details remain somewhat elusive, akin to the view from a camera obscura. We anticipate numerous modifications and refinements as the project evolves. Therefore, we aim for the source code to also function as a form of documentation during this developmental phase.

5.1.Walk through the code

The initial phase involves collecting emails from a specific day as part of a Proof of Concept (PoC). We are adopting a practical, hands-on methodology to simulate the processing of emails on a day-to-day basis.

For instance, on April 13, 1999, the process to extract emails for that day would look like the following in our system:

```
Python
"""
| 199 | 1999-04-09 |
| 200 | 1999-04-12 |
| 201 | 1999-04-13 |
| 202 | 1999-04-14 |
| 203 | 1999-04-15 |
| 204 | 1999-04-19 |
| 205 | 1999-04-20 |
| 206 | 1999-04-21 |
| 207 | 1999-04-22 |
| 208 | 1999-04-23 |
| 209 | 1999-04-26 |
| 210 | 1999-04-27 |
"""
date = "1999-04-13"
df_day = LisaMeta.get_emails_by_date(db_source, date, fields, 100)
```

Following the retrieval of email data, the next step involves integrating this data into the transactional 't_nodes' table:

Python

```
# Ingest the data into the store
columns_daily = LisaMeta.ingest_df_by_column(db_meta, df_day, "daily")
row_daily = LisaMeta.ingest_df_by_row(db_meta, df_day, "daily"; p=10)
```

The most critical phase is then initiated, where the system undergoes a regeneration process. This involves assimilating the changes within the System environment, brought about by the newly processed email data for another day:

Python

```
# Commit the data to the store
message = string("Ingested data for ", date)
Store.commit(db_meta, "hll_algebra.hdf5", "Alex Mylnikov",
"alexmy@lisa-park.com", message, Config())
```

As previously discussed, the commit function plays a crucial role in updating the system's current state (referred to as the "**head**") and relocating any modified nodes and edges to the archive section (known as the "**tail**"). In the existing setup, all archived data is directed to an HDF5 data store.

Below, we outline the structure of this HDF5 file:

Unset

```
🗂️ HDF5.File: (read-only) hll_algebra.hdf5
├─ 📂 11b4bd95-40f9-46b2-876b-26e1dcc1a160
│  └─ 📂 nodes
│     ├─ 📂 6927c490b1aac716b820616b2de8b7191ee9d68d
│     │  └─ 🔢 ["column"]
│     │     ├─ 🏷️ column_name
│     │     ├─ 🏷️ column_type
│     │     └─ 🏷️ commit_id
│     ├─ 📂 8342e461fb2cb55b121fca66cc8ca745ee2f72ae
│     │  └─ 🔢 ["column"]
│     │     ├─ 🏷️ column_name
│     │     ├─ 🏷️ column_type
│     │     └─ 🏷️ commit_id
. . . . .
. . . . .
```

```
├─ 📂 72490f8b-9080-4a62-a368-01c40d50cefd
│  └─ 📂 nodes
│     ├─ 📂 6927c490b1aac716b820616b2de8b7191ee9d68d
│     │  └─ 🔢 ["column"]
│     │     ├─ 🏷️ column_name
│     │     ├─ 🏷️ column_type
│     │     └─ 🏷️ commit_id
│     ├─ 📂 8342e461fb2cb55b121fca66cc8ca745ee2f72ae
│     │  └─ 🔢 ["column"]
│     │     ├─ 🏷️ column_name
│     │     ├─ 🏷️ column_type
│     │     └─ 🏷️ commit_id
. . . . .
```

The commit function encapsulates two pivotal steps:

1. `commit_nodes` and
2. `commit_edges`,

each playing a vital role in the system's adaptation and evolution based on the ingested email data.

Let's delve deeper into the functionality and importance of these functions.

5.2. Committing new nodes (commit_nodes function)

Below is the source code for the 'commit_node' function:

Python

```
function commit_node(db::Graph.DB, hdf5_filename::String, commit_id::String)
    # Get all nodes from t_nodes
    t_nodes = DBInterface.execute(db.sqlitedb, "SELECT * FROM t_nodes") |>
DataFrame

    for row in eachrow(t_nodes)
        t_sha1 = string(row.sha1)
        row = update_props!(row, commit_id)
        # Check if the node exists in nodes
        nodes = DBInterface.execute(db.sqlitedb, """SELECT * FROM nodes WHERE
sha1 ='$t_sha1'""") |> DataFrame
        if !isempty(nodes)
```

```
            node = nodes[1, :]
            # Compare the props fields
            if row.card != node.card || JSON3.read(row.props) !=
JSON3.read(node.props)
                # Remove diff edges (HEW, RET, DEL) for if they exist
                DBInterface.execute(db.sqlitedb, """DELETE FROM edges WHERE
target ='$t_sha1' AND r_type IN ('NEW', 'RET', 'DEL')""")
                # Remove the node from nodes
                DBInterface.execute(db.sqlitedb, """DELETE FROM nodes WHERE
sha1 ='$t_sha1'""")
                # Remove diff nodes (NEW, RET, DEL) for if they exist
                DBInterface.execute(db.sqlitedb, raw"DELETE FROM nodes WHERE
json_extract(props, '$.this')" * " = '$t_sha1'")
                # println("Exporting node: ", t_sha1)
                export_node(db, node, hdf5_filename)

                # Calculate difference between new and old version of the node
and
                # Generate 3 nodes (NEW, RET, and DEL) and 3 edges (NEW, RET,
and DEL)
                # to represent the difference
                #-------------------------------------------------

                Graph.node_diff(db, row, node)

                #-------------------------------------------------
                # Load node tp nodes

                dataset = JSON3.read(row.dataset, Vector{Int})
                props = JSON3.read(row.props, Dict{String, Any})
                labels = row.labels
                labels = JSON3.read(labels, Array{String, 1})
                g_node = Graph.Node(row.sha1, labels, row.d_sha1, row.card,
dataset, props)
                Graph.replace!(db, g_node, table_name="nodes")
            end
        end
        # Remove the node from t_nodes
        DBInterface.execute(db.sqlitedb, """DELETE FROM t_nodes WHERE sha1
='$t_sha1'""")
        # Delete the record from the nodes table
        DBInterface.execute(db.sqlitedb, "DELETE FROM assignments WHERE id =
'$t_sha1'")
```

```
    end
end
```

Although it appears to be a substantial amount of code, much of it is plumbing code intended for simplification at a later stage. This plumbing code establishes the necessary environment for the execution of the only one node operation, specifically the **Graph.node_diff(. . .)** function, as outlined in the HllSet Relational Algebra [3].

Python
```
# Calculate difference between new and old version of the node and
# Generate 3 nodes (NEW, RET, and DEL) and 3 edges (NEW, RET, and DEL)
# to represent the difference
#-------------------------------------------------

Graph.node_diff(db, row, node)
```

This graph node operation is responsible for managing all modifications in the graph database, including evaluating changes and generating new nodes to address them.

Python
```
"""
    Tis function calculates the difference between two states (hll_1 and hll_2)
    of the same node.
    Difference presented as a set of three nodes:

    1. N - New in hll_1
    2. R - Retained in hll_1 from hll_2
    3. D - Deleted from hll_1

    The main purpose of this function is to monitor changes in the
node.dataset.
"""
function node_diff(db::DB, x::DataFrameRow, y::DataFrameRow;
label::String="diff", p::Int64=10)
    # Check if the nodes have the same sha1
    if x.sha1 != y.sha1
        error("The nodes have different sha1")
    end
    sha1s = []
```

```
push!(sha1s, x.sha1)
# Restore nodes x and y
hll_1 = SetCore.HllSet{p}()
hll_1 = SetCore.restore(hll_1, JSON3.read(x.dataset, Vector{UInt64}))
hll_2 = SetCore.HllSet{p}()
hll_2 = SetCore.restore(hll_2, JSON3.read(y.dataset, Vector{UInt64}))

# To ensure the uniqueness of generated sha1s for N, R, and D
# we are going to combine sha1 and commit id of the hll_1
arr_x = Array{String, 1}(undef, 2)
arr_x[1] = x.sha1
props_x = JSON3.read(x.props, Config)
arr_x[2] = JSON3.write(x.labels)
sha1_commit_x = Util.sha1_union(arr_x)
arr_y = Array{String, 1}(undef, 2)
arr_y[1] = y.sha1
props_y = JSON3.read(y.props, Config)
arr_y[2] = props_y["commit_id"]
sha1_commit_y = Util.sha1_union(arr_y)
#================================
delta = SetCore.diff(hll_1, hll_2)
#================================
for (name, value) in pairs(delta)
    # Add new edge
    arr = Array{String, 1}(undef, 2)
    arr[1] = sha1_commit_x
    arr[2] = string(name)
    sha1 = Util.sha1_union(arr)
    props_e = Config()
    props_e["commit_id"] = props_x["commit_id"]
    edge = Edge(sha1, x.sha1, string(name), props_e)
    replace!(db, edge)
    # Add new node
    d_sha1 = SetCore.id(value)
    card = SetCore.count(value)
    dataset = SetCore.dump(value)
    props_n = Config()
    props_n["this"] = x.sha1
    props_n["this_card"] = x.card
    props_n["prev"] = y.sha1
    props_n["prev_card"] = y.card
    props_n["commit_id"] = props_x["commit_id"]
    node = Node(sha1, [string(name)], d_sha1, card, dataset, props_n)
    push!(sha1s, sha1)
```

```
        replace!(db, node)
    end
    return sha1s
end
```

Once again, we encounter numerous lines of code, but the central element is

```python
#================================
delta = SetCore.diff(hll_1, hll_2)
```

Below is the source code of this HllSet operation:

```python
function Base.diff(hll_1::HllSet{P}, hll_2::HllSet{P}) where {P}
    # x = count(intersect(hll_1, hll_2))
    # d = count(hll_1) - x
    # r = x
    # n = count(hll_2) - x
    # return (D = d, R = r, N = n)
    n = HllSet{P}()
    d = HllSet{P}()
    r = HllSet{P}()

    n = set_comp(hll_1, hll_2)
    d = set_comp(hll_2, hll_1)
    r = intersect(hll_1, hll_2)

    return (DEL = d, RET = r, NEW = n)
end
```

5.3. Committing new edges

Committing edges is a straightforward process since they are linked to nodes, and edges associated with previous versions of nodes remain unchanged. When new nodes are introduced, they require the creation of new edges, simplifying the record-keeping process.

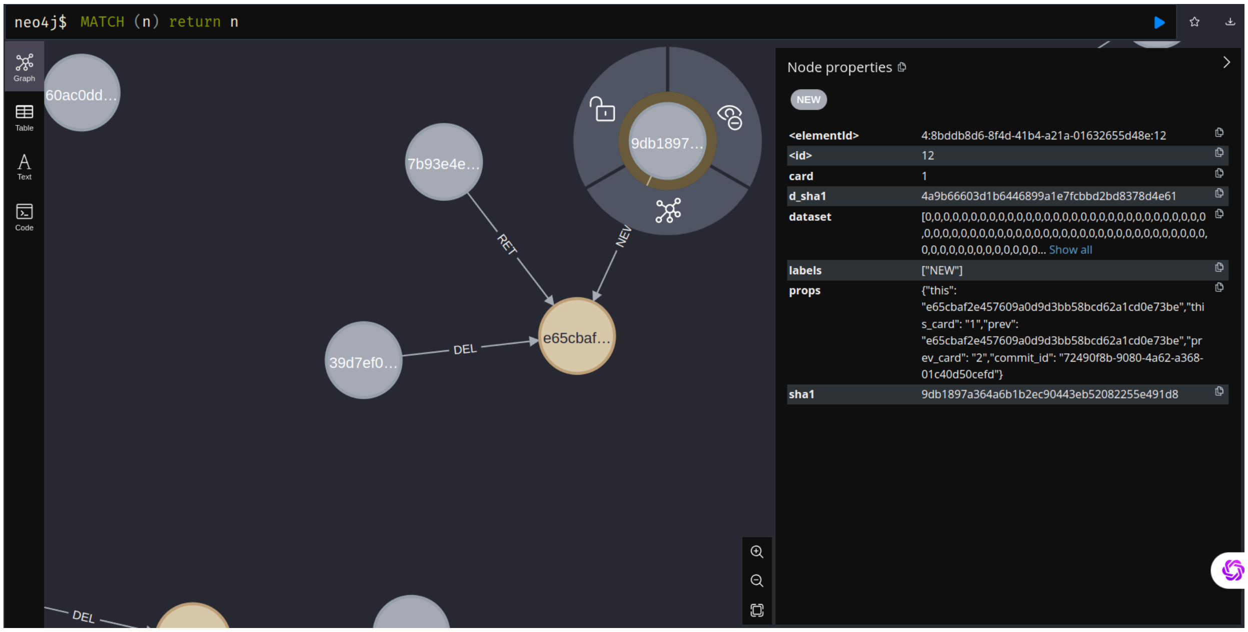
Here is the source code for committing the edges:

Python

```
function commit_edge(db::Graph.DB, hdf5_filename::String, commit_id::String)
    # Get all edges from t_edges
    t_edges = DBInterface.execute(db, "SELECT * FROM t_edges") |> DataFrame

    for row in eachrow(t_edges)
        t_source = string(row.source)
        t_target = string(row.target)
        t_type = string(row.r_type)

        row = update_props!(row, commit_id)
        # Check if the edge exists in edges
        edges = DBInterface.execute(db.sqlitedb, """SELECT * FROM edges WHERE
                source ='$t_source' AND target = '$t_target' AND r_type =
'$t_type'""") |> DataFrame
        if !isempty(edges)
            edge = edges[1, :]
            # Compare the props fields
            if JSON3.read(row.props) != JSON3.read(edge.props)
                export_edge(db, edge, hdf5_filename)
                # Remove the edge from edges
                DBInterface.execute(db.sqlitedb, """DELETE FROM edges WHERE
                        source ='$t_source' AND target = '$t_target' AND r_type
= '$t_type'""")
            end
        end
        # Remove the edge from t_edges
        DBInterface.execute(db.sqlitedb, """DELETE FROM t_edges WHERE
                source ='$t_source' AND target = '$t_target' AND r_type =
'$t_type'""")

        SQLite.load!(DataFrame(row), db.sqlitedb, "edges")
    end
end
```

5.4. Commit results in the Neo4J browser

This screenshot displays the modifications in the graph database, showcasing three new nodes in gray color. These nodes contain updates to the existing node depicted in orange, which represents the latest version of the column node.



Summary

The document discusses the concept of Self-Generative Systems (**SGS**), inspired by John von Neumann's theory of self-reproducing automata, and applies it to a Metadata Management System utilizing HyperLogLog Sets (**HllSets**) and graph databases to manage data and its metadata efficiently. It explores the structure and functions of self-replicating systems, which consist of modules for construction, copying, controlling, and interacting with the environment, along with a system description unit for storing module descriptions. The article outlines the implementation of these concepts through a proof of concept for a Metadatum SGS, which leverages Julia and Neo4J for metadata management.

The discussion includes the introduction of HllSets, which comply with set theory properties, and their mapping into a graph database structure, where nodes represent HllSets and edges represent connections between these sets. It delves into technical aspects like transactions, commits, and the handling of data in a self-generative manner, emphasizing the importance of commit statuses (Head, Tail, Deleted) in tracking modifications and maintaining the system's integrity.

Furthermore, the text distinguishes between static and dynamic metadata structures, highlighting the dynamic nature of data relationships and the role of analytical models in uncovering these relationships. A demonstration using data from the Enron Email Analysis project illustrates the practical application of these concepts, showcasing the process of ingesting, processing, and committing data to simulate the day-to-day operation of the Metadatum SGS.

Key functions such as `commit_node` and `commit_edge` are explained, detailing how they contribute to the system's ability to adapt and evolve by processing and integrating new data. The narrative emphasizes the potential of SGS in automating and standardizing the development cycle of software applications, from development through testing to production, thereby facilitating autonomous software development.

Overall, the article presents a comprehensive exploration of self-replicating systems and their application to metadata management, offering insights into the potential of these systems to revolutionize how we handle and analyze data in various domains.

=========

HllSets are employed to manage sets of data efficiently, particularly advantageous in handling large volumes of data with minimal memory usage. Mapping these sets into a graph database structure enables a clear representation of relationships between data points, making it easier to manage and analyze metadata.

Self-Generative Systems are designed based on the principles of self-replication and automation. The key components of such systems include:
- **Construction Module**: Responsible for building new system components.
- **Copying Module**: Enables the duplication of system components for redundancy and scalability.
- **Control Module**: Manages and regulates the operations within the system.
- **Interaction Module**: Facilitates communication and interaction with the external environment.
- **System Description Unit**: Acts as a repository for storing descriptions and specifications of the system modules.

These elements work in concert to allow the system to self-replicate, adapt, and evolve in response to environmental changes, thus embodying the characteristics of autonomous systems.

The system’s integrity and efficiency in handling data modifications are maintained through specific commit statuses:
- **Head**: The latest active version of data.
- **Tail**: The original version of data before any modifications.
- **Deleted**: Flagged data removed from active use but retained for historical integrity.

Functions like `commit_node` and `commit_edge` are crucial for updating the state of the system, ensuring that each transaction is processed and integrated seamlessly, allowing the system to adapt continuously to new data inputs.

The document highlights the distinction between static and dynamic metadata structures. While static structures are unchanging, dynamic structures evolve, offering a more flexible and comprehensive view of data relationships. This is particularly useful in analytical models that aim to unearth hidden patterns and connections in data.

The exploration of SGS in the context of metadata management opens up new avenues for autonomous software development, potentially revolutionizing the way software applications are developed, tested, and deployed. By automating these processes, SGS can significantly reduce manual overhead, enhance system reliability, and accelerate the software development lifecycle.

References

1. https://archive.org/details/theoryofselfrepr00vonn_0/page/74/mode/2up
2. https://en.wikipedia.org/wiki/Kozma_Prutkov
3. https://www.linkedin.com/posts/alex-mylnikov-5b037620_hllset-relational-algebra-activity-7199801896079945728-4_bI?utm_source=share&utm_medium=member_desktop
4. https://www.linkedin.com/posts/alex-mylnikov-5b037620_hyperloglog-based-approximation-for-very-activity-7191569868381380608-CocQ?utm_source=share&utm_medium=member_desktop
5. https://www.linkedin.com/posts/alex-mylnikov-5b037620_hllset-analytics-activity-7191854234538061825-z_ep?utm_source=share&utm_medium=member_desktop
6. https://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf
7. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/40671.pdf
8. https://github.com/alexmy21/lisa_meta/blob/main/lisa.ipynb
9. https://www.linkedin.com/posts/alex-mylnikov-5b037620_demo-application-enron-email-analysis-with-activity-7195832040548614145-5Ot5?utm_source=share&utm_medium=member_desktop
10. https://github.com/alexmy21/lisa_meta/blob/main/lisa_enron.ipynb
11. https://github.com/alexmy21/lisa_meta/blob/main/hll_algebra.ipynb
12. https://arxiv.org/pdf/2311.00537 (Machine Learning Without a Processor: Emergent Learning in a Nonlinear Electronic Metamaterial)
13. https://s3.amazonaws.com/arena-attachments/736945/19af465bc3fcf3c8d5249713cd586b28.pdf (Deep listening)
14. https://www.deeplistening.rpi.edu/deep-listening/

Some additional background references:

15. https://en.wikipedia.org/wiki/Von_Neumann_universal_constructorhttps://en.wikipedia.org/wiki/Markov_property
16. https://benjaminwhiteside.com/2020/10/25/markov-chains/
17. https://en.wikipedia.org/wiki/%CE%A3-algebra
18. https://github.com/jsvine/markovify?tab=readme-ov-file
19. https://arxiv.org/pdf/2110.00602.pdf
20. https://web.stanford.edu/~montanar/TEACHING/Stat310A/lnotes.pdf
21. https://arxiv.org/pdf/2004.05631.pdf (Tai-Danae Bradley. At the Interface of Algebra and Statistics. 2020)
22. https://www.math.colostate.edu/~renzo/teaching/Topology10/Notes.pdf
23. https://web.stanford.edu/~boyd/vmls/vmls-julia-companion.pdf

24. https://libcats.org/book/441034
25. https://www.nature.com/articles/37261.pdf (traveling waves application in biology)
26. https://en.wikipedia.org/wiki/Periodic_travelling_wave
27. https://medium.com/@jrosseruk/demystifying-gcns-a-step-by-step-guide-to-building-a-graph-convolutional-network-layer-in-pytorch-09bf2e788a51
28. https://medium.com/@jrosseruk/differentiable-graph-module-dgm-for-gcns-explanation-pytorch-implementation-part-1-3236c97a8b25
29. https://github.com/antferdom/GDL/tree/master
30. https://a-j.gitbook.io/geometric-deep-learning
31. https://jonathan-hui.medium.com/kernels-for-machine-learning-3f206efa9418

Quantum computing references

1. https://libarch.nmu.org.ua/bitstream/handle/GenofondUA/16957/9c9781fd84da8e0419973d60506c432a.pdf?sequence=1&isAllowed=y (THE LOGIC OF QUANTUM MECHANICS by GARRETT BIRKHOFF AND JOHN VON NEUMANN)
2. https://leimao.github.io/blog/Grover-Algorithm/
3. https://medium.com/@belal.db/quantum-deep-dive-grovers-search-algorithm-383dcd771fda
4. https://quantumcomputing.stackexchange.com/questions/2110/whats-the-point-of-grovers-algorithm-if-we-have-to-search-the-list-of-elements
5. https://leimao.github.io/downloads/blog/2020-07-26-Simon-Algorithm/lecture06.pdf

5iqxhi46szojbfz2ajc6mmhk