HllSet demo: Enron Emails Analysis

Introduction

This demo application serves as a simulation of a real-world system that manages the collection of emails within a corporation on a daily basis. It is designed to extract metadata from these emails and perform basic analysis to establish connections between emails and users based on the content of the emails as well as the information found in the "To" and "From" fields of the dataset.

The source data is provided in a CSV file format. To streamline the simulation process, we are converting the CSV file into an SQLite database.

When utilizing a graph database, a node will be created for each email within the system. These nodes will be organized by calendar days, with users categorized within each day. To represent an email as a node in the graph, the row from the table will be converted into an HllSet. Furthermore, individual graph nodes will be generated for each field (column) present in the table. The process of constructing metadata will involve extracting daily data from the "emails" table in a sequential manner.

Acquiring source data

To access the dataset, please click on the following link: [Enron Email Dataset](https://data.world/brianray/enron-email-dataset/workspace/file?filename=enron_05_17_2015_with_labels_v2.csv%2Fenron_05_17_2015_with_labels_v2.csv).

Download Details:
- Size: 1.2 GB
- Content: 517,481 emails

Instructions for Use:
1. Once downloaded, the CSV file should be converted into an SQLite table for easier manipulation and querying.
2. The database should be named `lisa_enron.db`. Feel free to choose a different name for the database file, but remember to adjust the database name in all code segments accordingly where `lisa_enron.db` is referenced.

The following Julia code converts email data from a CSV file into a SQLite database:

```python
# Read the CSV file into a DataFrame
```

```
df = CSV.read("/home/alexmy/JULIA/DATA/enron_emails.csv", DataFrame)
print(first(df, 10))
# Columns From and To present values as
frozenset({'phillip.allen@enron.com'}).
# We want to extract only email address from this set.
df.From = map(x -> ismissing(x) ? "" : (isnothing(match(r"'([^']*)'", x)) ? ""
: match(r"'([^']*)'", x).captures[1]), df.From)
df.To = map(x -> ismissing(x) ? "" : (isnothing(match(r"'([^']*)'", x)) ? "" :
match(r"'([^']*)'", x).captures[1]), df.To)
# Write the DataFrame to a new SQLite table
SQLite.load!(df, db, "emails")
```

Next, we need to confirm that the data has been successfully loaded by executing a simple query on the "emails" table.

Python

```
# Query the first 10 rows of the table
df = DBInterface.execute(db, "SELECT * FROM emails ORDER BY Date LIMIT 10") |>
DataFrame
# Print the DataFrame
print(df)
```

Generate a list of dates for conducting the manual simulation:

Python

```
# Create list of dates
df_dates = DBInterface.execute(db, "SELECT DISTINCT strftime('%Y-%m-%d', Date)
AS Date FROM emails ORDER BY Date") |> DataFrame
# Get the first 10 rows of the DataFrame
df_dates_10 = first(df_dates, 10)
# Print the first 10 rows
print(df_dates_10)
```

Here is the output from this run.

Unset

```
10×1 DataFrame
| Row | Date       |
|     | String     |
├─────┼────────────┤
| 1   | 1980-01-01 |
```

```
| 2   | 1986-04-26 |
| 3   | 1986-05-01 |
| 4   | 1997-01-01 |
| 5   | 1997-03-03 |
| 6   | 1997-03-05 |
| 7   | 1997-03-06 |
| 8   | 1997-03-07 |
| 9   | 1997-03-11 |
| 10  | 1997-03-16 |
```

Enron Emails application in terms of General Statistics Theory

The Enron emails application can be described in terms of the general theory of statistics presented in the document Introduction to General Theory of Statistics. The application utilizes the concept of HllSet to collect metadata from emails and analyze the data both structurally and semantically.

HllSet is a probabilistic data structure that is used to estimate the number of distinct elements in a set with a high degree of accuracy. In the context of the Enron emails application, HllSet is used to collect metadata such as sender, recipient, timestamp, and subject of the emails. This metadata is then used to perform structural analysis by identifying patterns in the communication network, such as who is communicating with whom and how frequently.

Additionally, the metadata collected using HllSet can be used to perform semantical analysis of the emails. This involves analyzing the content of the emails to identify key topics, sentiments, and trends in the communication. By applying statistical techniques to the metadata collected from the emails, the Enron emails application can provide insights into the communication patterns and relationships within the organization.

Overall, the Enron emails application demonstrates how the general theory of statistics can be applied to analyze and extract valuable information from large datasets, such as email communications. By utilizing the concept of HllSet to collect metadata and perform both structural and semantic analysis, the application can help uncover hidden patterns and relationships within the data.

Daily Email Acquisition Simulation

We will be simulating the daily collection of emails by utilizing a series of specialized functions designed for this specific task.

Python

```
# Getting emails by date
function get_emails_by_date(date)
    return DBInterface.execute(db, "SELECT * FROM emails WHERE Date LIKE
'$date%'") |> DataFrame
end

# Ingesting email data into the store column by column
function ingest_df_by_column(db::Graph.DB, df::DataFrame, parent_sha1::String;
p::Int = 10)
    # Create a new HLL set
    hll = SetCore.HllSet{p}()
    for col_name in names(df)
        col = df[!, col_name]
        col_props = Config()
        col_props["column_name"] = col_name
        col_props["column_type"] = eltype(col)
        # col_props["parent_sha1"] = parent_sha1
        node_sha1 = Store.sha1(string(col_name, col_props["column_type"],
parent_sha1))
        col_hll = Store.ingest_dataset(db, Set(col), node_sha1, ["column"],
col_props, p)
        # Accumulate the column into the HLL set for current day
        hll = SetCore.union!(hll, col_hll)
    end
    return hll
end

# Ingesting email data into the store row by row
function ingest_df_by_row(db::Graph.DB, df::DataFrame, parent_sha1::String;
p::Int = 10)
    # Create a new HLL set
    hll = SetCore.HllSet{p}()
    for i in 1:size(df, 1)
        row = df[i, :]
        row_props = Config()
        # row_props["Message_ID"] = df[i, :Message-ID]
        # row_props["parent_sha1"] = parent_sha1
        row_props["Date"] = df[i, :Date]
        row_props["From"] = df[i, :From]
        row_props["To"] = df[i, :To]
        node_sha1 = Store.sha1(string(df[i, :Message-ID], row_props["Date"],
row_props["From"], row_props["To"]))
        row_hll = Store.ingest_dataset(db, Set(row), node_sha1, ["row"],
row_props, p)
```

```
        # Accumulate the row into the HLL set for current day
        hll = SetCore.union!(hll, row_hll)
    end
    return hll
end

function select_nodes_by_label(db::Graph.DB, label::String, table::String,
n::Int)
    refs = Set()
    query = ""
    if n == -1
        query = "SELECT * FROM $table WHERE label = '$label'"
    else
        query = "SELECT * FROM $table WHERE label = '$label' ORDER BY RANDOM()
LIMIT $n"
    end

    results = DBInterface.execute(db, query) |> DataFrame
    for result in eachrow(results)
        if length(result) > 0
            push!(edges, result)
            refs = union(refs, Set([result.sha1]))
        end
    end
    return refs
end
```

Obtaining daily email data

We will initiate the simulation in "manual" mode by executing the get_emails_by_date() function, using a hardcoded date as input.

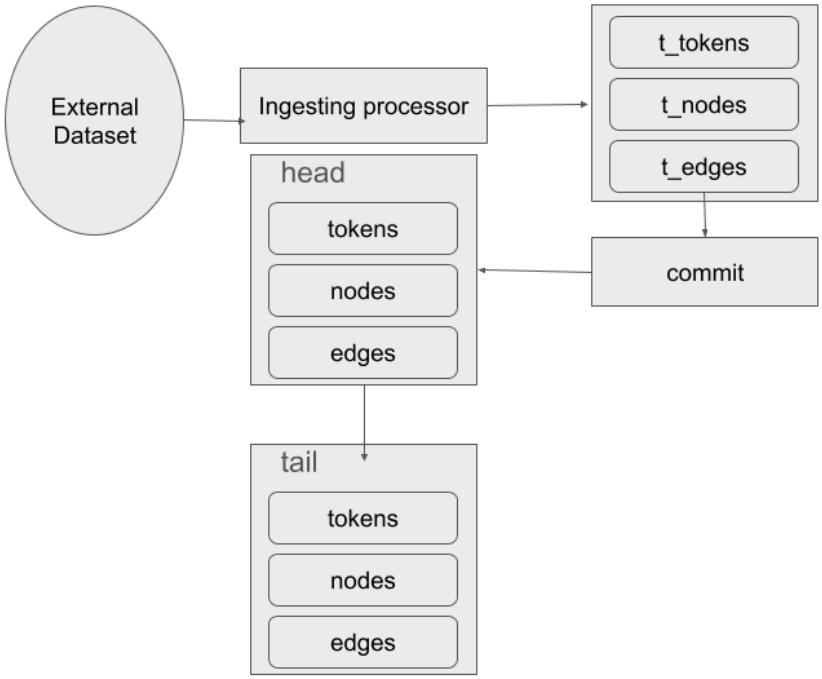
Python

```
df_day = get_emails_by_date("1980-01-01")
# Get the first 10 rows of the DataFrame
df_first_10 = first(df_day, 10)
# Print the first 10 rows
print(df_first_10)
```

Ingesting daily email data

The following steps outline the daily email data ingestion process:

1. Data is ingested into temporary storage, which only contains new data for the current day.
2. The ingested data is committed to the head of the metadata store.
3. If the head of the metadata store already contains an ingested node or edge with the same SHA1 ID but a different cardinality, the old version will be moved to tail storage for archiving purposes.



The following code is designed to carry out data ingestion:

```python
# Ingest the data into the store
columns_daily = ingest_df_by_column(db, df_day, "daily")
row_daily = ingest_df_by_row(db, df_day, "daily")

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

The output generated by this code encompasses two key components:
- `columns_daily`, an HllSet that amalgamates the HllSets from every column within the email dataset, offering a consolidated view of columnar data.
- `row_daily`, which represents a unified HllSet that aggregates the HllSets across each individual record (row) in the email dataset, providing a holistic snapshot of row-wise data.

Additionally, the `Store.commit()` function plays a crucial role in migrating data from temporary tables (`t_tokens`, `t_nodes`, and `t_edges`) to their designated permanent counterparts (`tokens`, `nodes`, and `edges`). This process ensures the seamless transition of data from a provisional to a stable storage state.

Discovering Enron Email Dataset structure

We now possess a collection of HllSets that capture all emails gathered on a particular day. As previously noted, an HllSet functions as a set, allowing us to utilize various set operations to establish connections between the emails.

It is important to remember that each entry in the nodes table corresponds to an instance of the Graph.Node struct.

```
Python
struct Node <: AbstractGraphType
       sha1::String
       labels::Vector{String}
       d_sha1::String
       card::Int
       dataset::Vector{Int}
       props::Config
end
```

In this structure, we have the following components:
- `sha1`: This is a SHA1 hash that acts as a unique identifier for the node.
- `labels`: These are graph labels assigned to the node.
- `d_sha1`: Another SHA1 hash, which serves as the identifier for the HllSet representation of the original dataset.
- `dataset`: This is a compressed version of the HllSet.
- `props`: A dictionary utilized for storing additional attributes.

Among these elements, the `dataset` and `props` are particularly instrumental in uncovering latent relationships within email data. The `props` attribute, a collection of additional properties, includes crucial fields such as From, To, and Date. This enables us to aggregate emails according to the senders, recipients, or the dates of the emails.

The aggregation process using `props` involves three main steps:

1. Generate a list of unique values for a specific property, such as the sender's email address in the 'From' field.
2. Retrieve all relevant HllSets from the `dataset` attribute and combine them into a unified HllSet, named as `from_hll_`, for each sender.
3. Establish connections between individual emails and the aggregated HllSet, `from_hll_`, effectively creating edges in the graph.

Additionally, there's an alternative method based purely on sets to gather related HllSets. This method involves creating an HllSet that contains a single element (but with two tokens) representing the value from the 'From' field. This approach provides another avenue to analyze and understand the relationships within the dataset.

```python
hll_from = SetCore.HllSet{10}()
# email address in the HllSet is tokenized. So,
'phillip.allen@enron.com'
# would be presented in HllSet as Set('phillip', 'allen', 'enron',
'com')
email = Set('phillip', 'allen')
SetCore.add!(hll_from, email)
```

Let's say hll_daily is a collection of all HllSets obtained from a row-based presentation of all emails collected on a particular day. In this scenario, the aggregation of emails belonged to a specific sender can be represented by the following pseudocode:

```python
# Initialize the HllSets
hll_all = Set{SetCore.HllSet{10}}()           # Collection of row HllSets from
hll_result = Set{SetCore.HllSet{10}}()        # Collection of resulting HllSets

hll_match = SetCore.HllSet{10}()        # Matching HllSet
hll_from_column = SetCore.HllSet{10}()  # HllSet that represents column with
name 'From'


# Populate the HllSets with some data
# ...
match_card = SetCore.count(hll_match)

# Find the HllSets in hll_all that have a non-empty intersection with
hll_from_column
# and hll_match with the cardinality of the intersection that is equal 2
for hll in hll_all
    # First extract from the row only values from column 'From'
    intersection = SetCore.intersect(hll, hll_from_column)
    # Match extracted values with hll_match
    intersection = SetCore.intersect(intersection, hll_match)
```

```
    if !SetCore.isempty(intersection) && SetCore.count(intersection) == 2
        SetCore.add!(hll_result, hll)
    end
end

# Print the result
println(hll_result)
```

This example subtly incorporates the concept of presenting data in a tabular format using HllSet. Essentially, we construct a single cell within the table for a designated row and column, aligning it with the `hll_match` HllSet to serve as a matching criterion.

To elucidate, we showcase how to leverage `SetCore.intersect(hll, hll_from_column)` for extracting information from rows under the "From" column. Following this, we apply `SetCore.intersect(intersection, hll_match)` to identify the overlap between the `intersection` and `hll_match` sets. To check whether the intersection is devoid of elements, the `SetCore.isempty(intersection)` function comes into play.

Moreover, the `SetCore.count(intersection)` function is utilized to determine the intersection's cardinality—that is, its element count. Under specific conditions, `SetCore.add!` is employed to incorporate `hll` into `hll_result`.

It's crucial to note that this example presumes the HllSets have been correctly initialized in advance.

However, the proposed scenario of discovering data structure may be considered a bit overly complex from a practical standpoint. A simpler approach would be to transfer all nodes into a Graph Database such as Neo4J and utilize the Cypher query language to manipulate nodes and establish desired relationships between them. This is the strategy we will be implementing in the following section as we explore using Neo4J for discovering relationships within the Enron Email Dataset.

Using Neo4J for discovering relationships within the Enron Email Dataset

First, we will transfer all nodes to the Neo4J database. This is achieved by employing the `select_nodes_by_label()` function. The final parameter of this function specifies the size of a random sample, where a value of `-1` indicates that all nodes are being extracted.

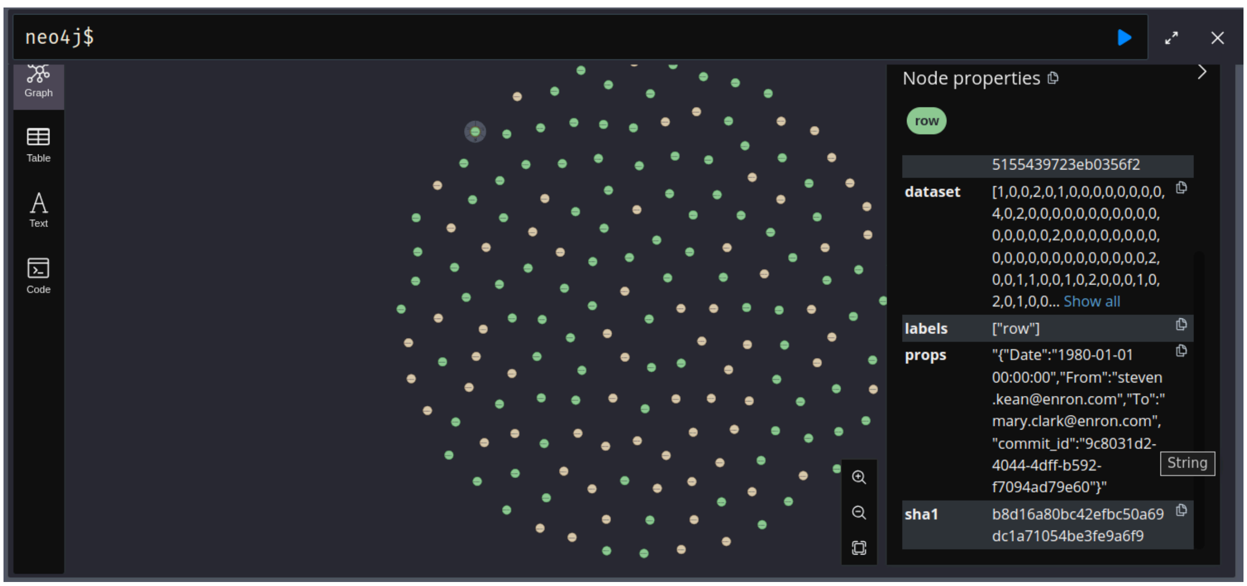
We are retrieving all nodes labeled as `column`, as well as selecting only 100 nodes labeled as `row`.

Python
```
refs_col = select_nodes_by_label(db_meta, "column", "nodes", -1)
refs_rows = select_nodes_by_label(db_meta, "row", "nodes", 100)
refs = union(refs_col, refs_rows)

nodes = Vector()
LisaNeo4j.select_nodes(db_meta.sqlitedb, refs, nodes)

# Add nodes to the Neo4j database
for node in nodes
    labels = replace(string(node.labels), ";" => "")
    query = LisaNeo4j.add_neo4j_node(labels, node)
    data = LisaNeo4j.request(url, headers, query)
end
```

The output generated by running this code is as follows:

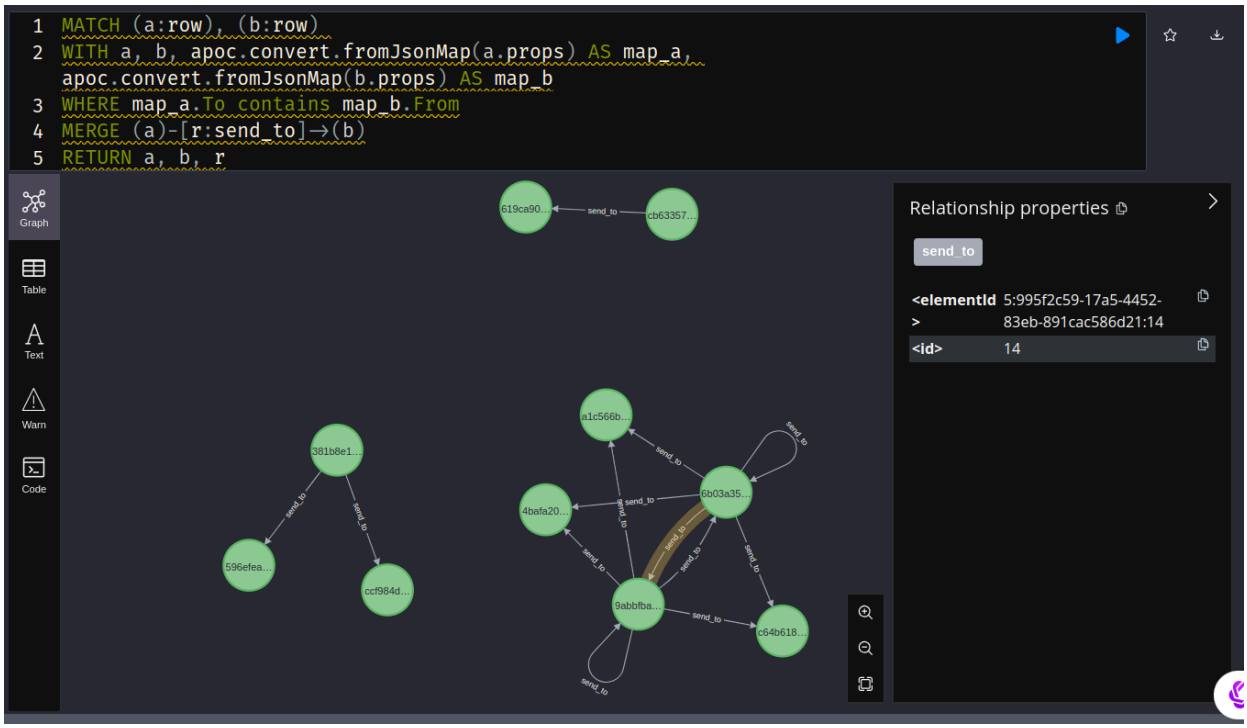


Once all nodes have been migrated to Neo4J, we have the opportunity to leverage the capabilities of the Cypher Query Language for the majority of tasks associated with analyzing the structure of email data. Here's an illustration of how to establish relationships between connected emails, tracing from the sender to the recipients.

We are utilizing the following Cypher query to create it.

```python
MATCH (a:row), (b:row)
WITH a, b, apoc.convert.fromJsonMap(a.props) AS map_a,
apoc.convert.fromJsonMap(b.props) AS map_b
WHERE map_a.To contains map_b.From
MERGE (a)-[r:send_to]->(b)
RETURN a, b, r
```

Here is a screenshot from the Neo4J browser, showcasing a Cypher query and the resulting graph.



Similarly, we have the ability to create additional aggregation relationships between email nodes, such as aggregating all emails associated with a particular user. However, Cypher does not currently have native support for operations on HllSets. This limitation is understandable, as HllSets are not yet widely known in the Cypher community. Therefore, the most effective approach may be a hybrid solution that combines traditional graph manipulation techniques with a programmatic approach to building an analytical graph.

The provided Julia code snippet calculates similarity relationships between emails based on the content similarity. It employs Jaccard and cosine distances as the metric for assessing this relationship.

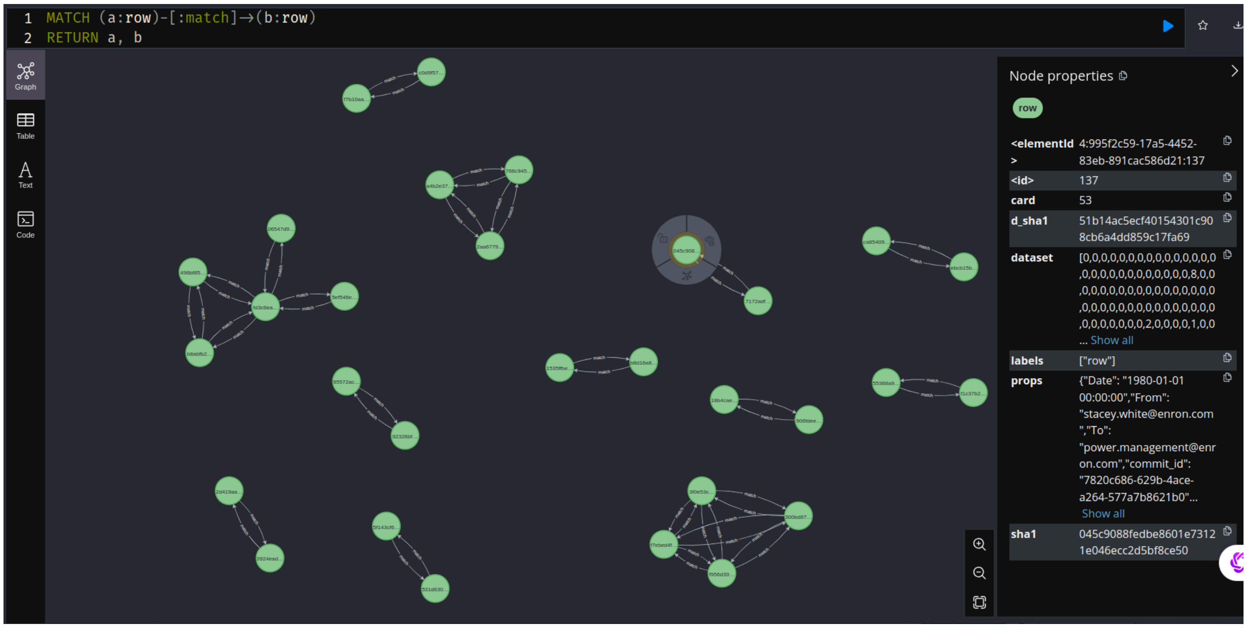
In the code the threshold is set to 50%.

Python

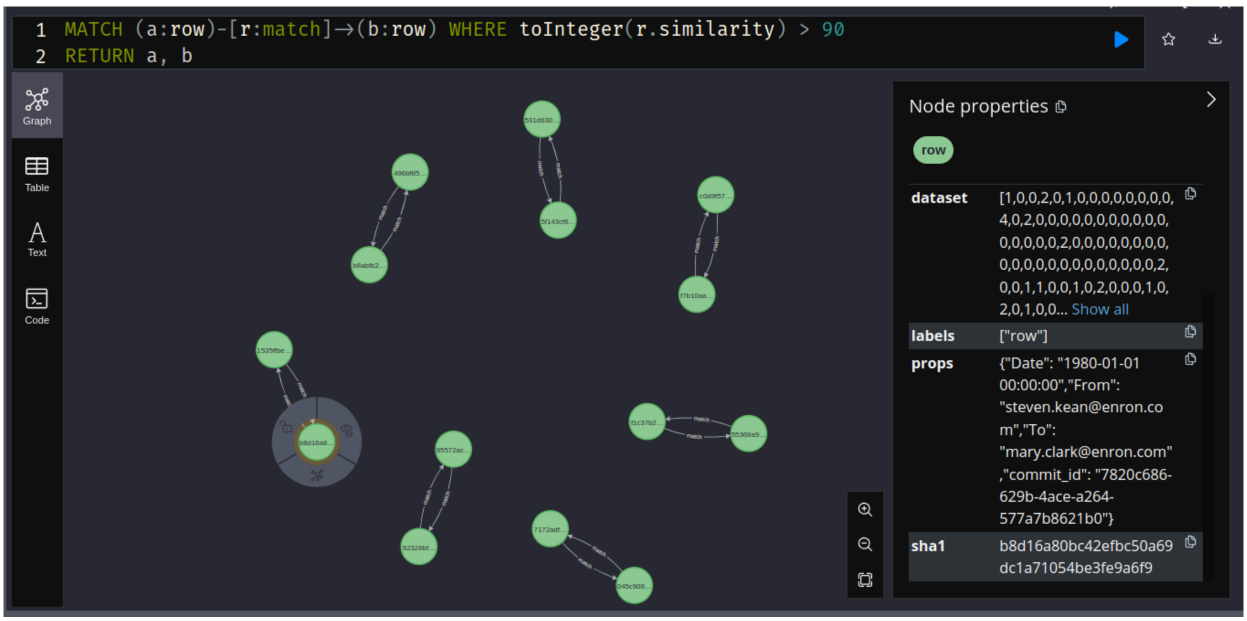
```
# Define your Cypher query
query = LisaNeo4j.cypher("MATCH (n:row) RETURN n.labels, n.sha1, n.d_sha1,
n.dataset, n.props")
# Parse the response
json = LisaNeo4j.request(url, headers, query)

hlls = Dict{String, LisaNeo4j.Neo_node}()
LisaNeo4j.collect_hll_sets(json, hlls)
# Generate the relations between matched emails
for (k, v) in hlls
    for (k1, v1) in hlls
        if k != k1
            match = SetCore.match(v.hll_set, v1.hll_set)
            cosine = SetCore.cosine(v.hll_set, v1.hll_set)
            if match > 50
                props = JSON3.write(Dict("similarity" => match))
                dict = Dict{String, Any}("source" => k, "target" => k1,
"r_type" => "match", "props" => props)
                df_row = LisaNeo4j.dict_to_dfrow(dict)
                query = LisaNeo4j.add_neo4j_edge(df_row)
                data = LisaNeo4j.request(url, headers, query)
            end
        end
    end
end
```

Graph illustrating the relationship among emails with a Jaccard distance exceeding 50%.



Here are all the emails that have a similarity score of over 90% according to the Jaccard distance.



Summary

The text describes a demo application called HllSet Enron Email Analytics that simulates a system for managing emails in a corporation. It extracts metadata from emails and analyzes connections between emails and users based on content and email fields.

The source data is provided in CSV format and converted into an SQLite database for analysis.

The application utilizes HllSet, a probabilistic data structure, to collect metadata and perform structural and semantic analysis on the Enron email dataset.

The text also outlines steps for acquiring, ingesting, and analyzing daily email data, as well as discovering relationships within the dataset using HllSets.

Additionally, it discusses using Neo4J and Cypher Query Language for further analysis and relationship discovery. The text concludes with a code snippet demonstrating similarity relationships between emails based on content similarity.