HllSet Relational Algebra

Disclaimer:

Please note that this document does not aspire to meet the standards of a formal mathematical paper. It has been authored by an individual without professional or amateur status in the field of mathematics. The purpose of this work is to share a collection of ideas that I found intriguing during my engagement with HllSets, in the hope that they might capture the interest of mathematicians who could potentially transform them into a significant mathematical discourse.

Introduction

In the Wikipedia entry [7], Relational Algebra is characterized as follows: "Relational algebra, in the realm of database theory, is a theoretical framework that employs algebraic structures to model data and formulate queries based on rigorously defined semantics. This theory was initially proposed by Edgar F. Codd. [8]"

In our paper, we aim to explore the concept of relational algebra through the lens of HllSets. It should be noted that this document is in its preliminary stages, and extensive development is anticipated.

HllSets

In a previous post [1], I introduced HllSets, a data structure based on the HyperLogLog algorithm developed by Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier [3]. 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. $(A \cup B) = (B \cup A)$
2. $(A \cap B) = (B \cap A)$

Associative Property:
3. $(A \cup B) \cup C) = (A \cup (B \cup C))$
4. $(A \cap B) \cap C) = (A \cap (B \cap C))$

Distributive Property:
5. $((A \cup B) \cap C) = (A \cap C) \cup (B \cap C)$
6. $((A \cap B) \cup C) = (A \cup C) \cap (B \cup C)$

Identity:
7. $(A \cup \varnothing) = A$

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**[5].

Mapping HllSets into Graph DB

The graph can be defined as follows:

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
```

This structure is standard for representing any graph node. The set of node details is minimal and serves to ensure unambiguous identification of each node of the graph.

In this structure we distinguish three parts:

1. The **external** description of the node, which, in turn, is represented by two attributes:
   a. **sha1** - unique identifier (SHA1 hash);
   b. **labels** - labels (one or more) that reflect the semantics of a given node;
2. **Internal** description of a node, which describes the content of the CUI represented by this node. Attributes of this description include:
   a. **d_sha1** - SHA1 hash generated from the contents of the SEI;
   b. **dataset** - vectorized representation of the contents of the CUI (we will dwell on the vector representation of the CUI in more detail in the next section);
   c. **card** - the number of non-repeating dataset elements;
3. **props** - additional node attributes, presented as a JSON structure.

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.

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
```

Edge description attributes:

1. **'source'** - sha1 identifier of the starting node;
2. **'target'** - sha1 end node identifier;
3. **'r_type'** - edge label reflecting the type of connection between nodes;
4. **'props'** - additional edge attributes, presented as a JSON structure.

As you can see the graph by itself doesn't provide any reference to real values encoded in node elements of the real datasets. To address this issue we implemented a vocabulary that contains all tokens ingested into the system.

The system vocabulary, or dictionary, encompasses all collected values for attributes. This collection of terms is dynamic, constantly growing to incorporate new concepts and terms as the observed reality evolved.

To describe the system dictionary we use the following structure:

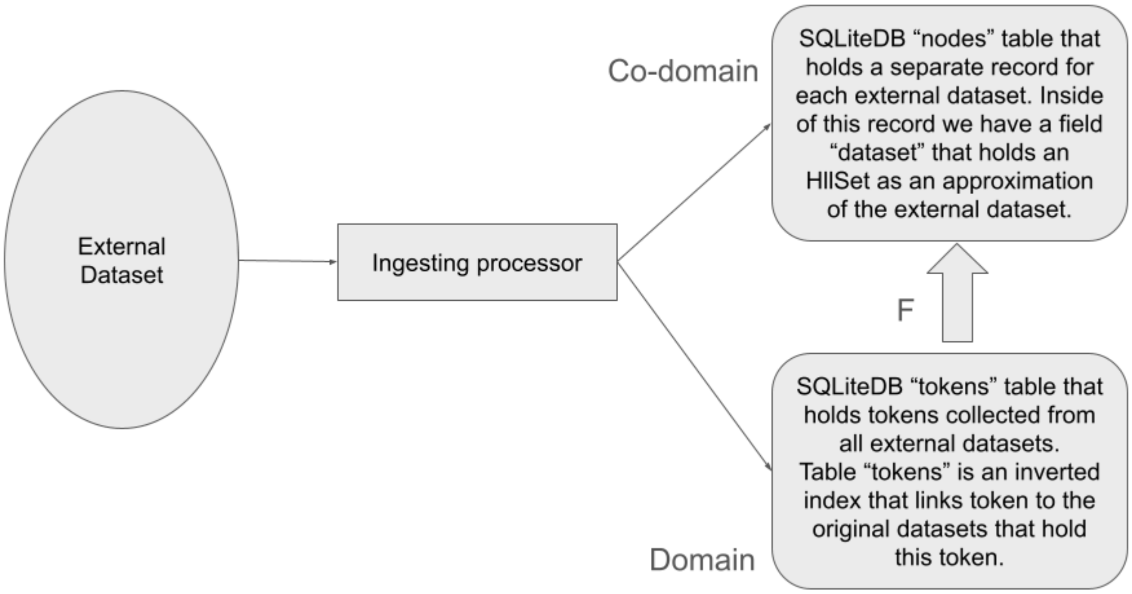
Python

```
struct Token <: AbstractGraphType
        id::Int           # UInt64 hash value
        bin::Int            # Integer generated fro p leading bits of "id" field
        zeros::Int          # Number of tailing zeros in th hash id
        token::Set{String} # string presentation dataset element
        tf::Int             # term frequency number of references for the token
        refs::Set{String}  # Disting collection of references for the token
end
```

In this structure:

1. `**id**`: An integer generated by hashing the value of the word (token). A token refers to any code that signifies a basic unit of data, regardless of its data type.
2. `**bin**`: The first k bits of the hash code, indicating the token's position in the HllSet representation of the dataset.
3. `zeros`: The count of consecutive zeros observed at the end of the hash's bit representation of the tokens.
4. `**tf**`: The term frequency, which represents how often a particular token occurs within the datasets.
5. `**refs**`: A collection of sha1 identifiers representing all the HllSets that include this word.

The diagram below illustrates the communication process between processors that ingest real data, the system dictionary, and nodes that represent the metadata of the collected data.



SQLiteDB “nodes” table that holds a separate record for each external dataset. Inside of this record we have a field “dataset” that holds an HllSet as an approximation of the external dataset.

SQLiteDB “tokens” table that holds tokens collected from all external datasets. Table “tokens” is an inverted index that links token to the original datasets that hold this token.

Identity problem

There are only two hard things in Computer Science: cache invalidation and naming things.
-- Phil Karlton

You probably noticed that strut Node has 2 identifiers for each node:

- sha1
- d_sha1.

The first one we call **external identifiers** and the second is **internal**.

In 1970 (or even earlier), the concept of using an address value as an ID was first implemented in Content Addressable Storage (CAS) [9].

Below is an excerpt from a reference [9] discussing the use of the ISBN system to assign a unique number to each book.

> “By searching for the ISBN 0465048994 online, one can find various locations where the book "Why Information Grows" is available, all referring to the same work on information storage. This allows users to choose the most suitable location for accessing the content. Furthermore, in the event that a particular location is altered or removed, the content can still be accessed from other locations.

A modern and prominent example of CAS implementation is the Git Data Store. Git operates as a content-addressable filesystem, meaning that it utilizes a simple key-value data store at its core. This allows users to insert various types of content into a Git repository, with Git providing a unique key for later retrieval of that content.

Git employs a SHA1 hash that is created from the file content. **Taking a cue from Git, we have implemented a similar approach by generating a SHA1 hash from a set of specific properties extracted from the metadata that defines the datase**t. But why do we not use the entire instance of the dataset to calculate the SHA1 hash?

This question delves into a philosophical aspect. Consider this scenario: if I measure my weight before lunch and calculate the SHA1 hash with weight as one of the defining properties of my entity instance, the resulting SHA1 ID would be unique. However, if I were to calculate the SHA1 ID after lunch, it would likely be different.

This implies that, according to the SHA1 hash, the person before lunch and after lunch are not the same, which is not the intended outcome as they are essentially identical.

The solution to this philosophical dilemma is straightforward - avoid using frequently changing properties as the basis for calculating unique IDs using SHA1 or any other hash functions.

On the other hand, we aim to maintain a unique ID that represents the entire HllSet at any point when we are calculating SHA1 hash. To achieve this, we introduced an additional **internal ID**, denoted as **d_sha1**, within the Node struct. We utilize this secondary ID to track internal variations within the observed dataset by assessing the modifications determined for the HllSet, which approximates the given dataset.

There are two main types of HllSets: **original HllSets**, which are created directly from the original datasets, and **generated HllSets**, which are created by applying set operations to other HllSets. The key distinction between them lies in the method used to calculate the sha1 ID of the node that represents the HllSet.

Calculating d_sha1 for HllSet

As previously discussed, the process involves computing the **d_sha1** value by converting the entire HllSet into a SHA1 string. Below is the code snippet that executes this calculation:

Python

```
module SetCore
   . . . . .
   function id(x::HllSet{P}) where {P}
        # Convert the Vector{BitVector} to a byte array
        bytearray = UInt8[]
        for bv in x.counts
            append!(bytearray, reinterpret(UInt8, bv))
        end
        # Calculate the SHA1 hash
        hash_value = SHA.sha1(bytearray)
        return SHA.bytes2hex(hash_value)
    end
    . . . . .
end

d_sha1 = SetCore.id(hll)
```

The output of **d_sha1** is solely determined by the HllSet that is provided, as demonstrated in the code.

Calculating sha1 for original HllSets

In the scenario of **original** HllSets, the **sha1** ID mirrors the essence of the original dataset. This mirroring process is handled directly within the code of a specialized processor designed for ingesting datasets. These processors are specifically developed and optimized for various dataset types, including SQL databases, CSV files, documents, images, video files, among others.

In the code snippet we are providing example of calculating **sha1** ID for the dataset that represent a column from csv file:

```python
# sha1 is a function that convert a string into SHA1 hash
# bytes2hex is a function that conver SHA1 hash into string

sha_1 = bytes2hex(sha1(file_name * column_name * column_type))
```

Calculating sha1 for generated HllSets

One might naturally anticipate that the SHA-1 hash of the resulting HllSet, derived from operations on other HllSets, would be a function of the SHA-1 hashes of the operand HllSets.

The primary criterion for the SHA-1 calculation algorithm is that it adheres to the fundamental properties associated with HllSets. This implies that the algorithm should validate the following assertions:

Commutative Property:

1. $\text{sha1}\ (A \cup B) = \text{sha1}(B \cup A)$
2. $\text{sha1}\ (A \cap B) = \text{sha1}(B \cap A)$

Associative Property:

3. $\text{sha1}((A \cup B) \cup C) = \text{sha1}((A \cup (B \cup C))$
4. $\text{sha1}(\ (A \cap B) \cap C) = \text{sha1}((A \cap (B \cap C))$

Distributive Property:

5. $\text{sha1}\ ((A \cup B) \cap C) = \text{sha1}(\ (A \cap C) \cup (B \cap C))$
6. $\text{sha1}((A \cap B) \cup C) = \text{sha1}(\ (A \cup C) \cap (B \cup C))$

Identity:

7. $\text{sha1}(A \cup \varnothing) = \text{sha1}(A)$
8. $\text{sha1}(A \cap U) = A$

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

Idempotent Laws:

9. $\text{sha1}(A \cup A) = \text{sha1}(A)$
10. $\text{sha1}\ (A \cap U) = \text{sha1}(A)$

Anti Commutative property for relative complement operation:

11. $\text{sha1}(A \setminus B) \neq \text{sha1}(B \setminus A)$, except of trivial case $A = B$

Before delving into the implementation of the sha1 algorithm for the HllSet operation, we would like to introduce some utility functions:

JavaScript

```
function char_to_bin(c::Char)
    return string(UInt8(c), base=2)
end

function string_to_bin(str)
    return join([char_to_bin(c) for c in str])
end

function bin_to_string(bin_str)
    return join([Char(parse(UInt8, bin_str[i:min(i+7, end)] , base=2)) for i in
1:8:length(bin_str)])
end
```

sha1_union

Python

```
# strings argument is an array of HllSets sha1 IDs that partisipate in union
operation
function sha1_union(strings::Array{String, 1})
    bin_strings = [string_to_bin(str) for str in strings]
    bin_union = bin_strings[1]
    for i in 2:length(bin_strings)
        bin_union = string(parse(BigInt, "0b" * bin_union) | parse(BigInt, "0b"
* bin_strings[i]), base=2)
    end
    str_union = bin_to_string(bin_union)
    new_sha1_hash = bytes2hex(SHA.sha1(str_union))

    return new_sha1_hash
end
```

sha1_intersection

Python

```
# strings argument is an array of HllSets sha1 IDs that partisipate in intersect
# operation
function sha1_intersection(strings::Array{String, 1})
    bin_strings = [string_to_bin(str) for str in strings]
    bin_intersection = bin_strings[1]
    for i in 2:length(bin_strings)
        bin_intersection = string(parse(BigInt, "0b" * bin_intersection) & parse(BigInt, "0b" * bin_strings[i]), base=2)
    end
    str_intersection = bin_to_string(bin_intersection)
    new_sha1_hash = bytes2hex(SHA.sha1(str_intersection))

    return new_sha1_hash
end
```

sha1_complement

Python

```
# strings argument is an array of HllSets sha1 IDs that partisipate in
# complement operation
function sha1_complement(sha_1::String, sha_2::String)
    bin_1 = string_to_bin(sha_1)
    bin_2 = string_to_bin(sha_2)
    bin_complement = string(parse(BigInt, "0b" * bin_1) & ~parse(BigInt, "0b" * bin_2), base=2)
    str_complement = bin_to_string(bin_complement)
    new_sha1_hash = bytes2hex(SHA.sha1(str_complement))

    return new_sha1_hash
end
```

Testing sha1_ algorithms

Python

```
strings = Set{String}()
push!(strings, "aba567b86136d0ad7cdaec68e600ca02a3400544")
push!(strings, "g1a567b86136d0ad7cdaec68e600ca02a3400530")
```

```
push!(strings, "b1a567b86136d0ad7cdaec68e600ca02a3400526")

println("sha1_union([A, B, C]): ", sha1_union(strings))
println("sha1_intersect([A, B]): ",
sha1_intersect("g1a567b86136d0ad7cdaec68e600ca02a3400530",
"aba567b86136d0ad7cdaec68e600ca02a3400544"))

println("sha1_complement(B, A): ",
sha1_comp("g1a567b86136d0ad7cdaec68e600ca02a3400530",
"aba567b86136d0ad7cdaec68e600ca02a3400544"))

println("sha1_complement(A, B): ",
sha1_comp("aba567b86136d0ad7cdaec68e600ca02a3400544",
"g1a567b86136d0ad7cdaec68e600ca02a3400530"))
```

Here is the output produced by the preceding code.

Unset

```
sha1_union([A, B, C]): c9847d0263de186fc4e3f56b648895565bcd5952
sha1_intersect([A, B]): a899d4a28e2c26ae888e964aebec6bfedf30b806
sha1_comp(B, A): e08b92300f215b925bca6dd3719ae76110c5279d
sha1_comp(A, B): ff568d861f973009bffc59bea505aa9305a70b1e
```

The last two lines display the results of the sha1_comp() function for the same arguments, but in a different order, demonstrating that the outputs are distinct.

Operations on HllSet Nodes

We'd like to make it clear from the outset that any operation performed on HllSet nodes will result in the creation of a new HllSet node. This new node may expand the existing collection of nodes. Therefore, it's important to ensure that the newly created node is effectively integrated into the Graph DB.

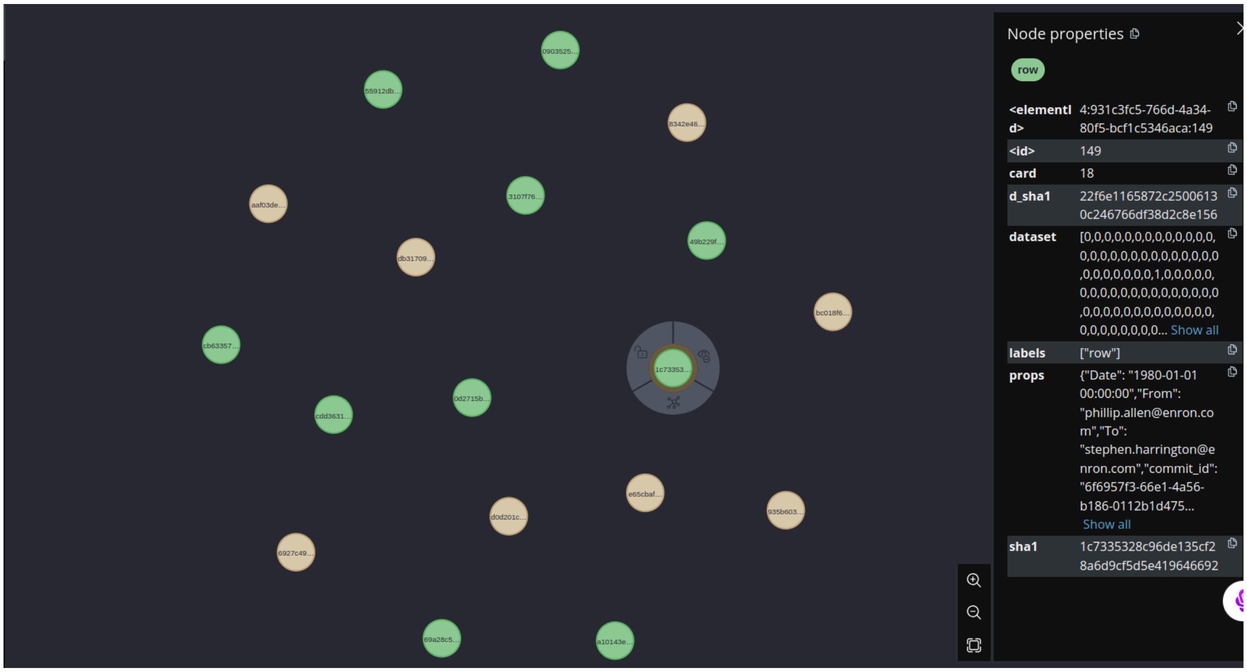
Every operation will impact two primary tables in the Graph DB for the following reasons:

1. Table “**nodes”** needs to be updated with the newly created node.
2. Table “**edges”** should be updated with new edges that will connect the new node with the HllSet nodes used as arguments in the operation.

These modifications should be made in the staging version of these tables (prefixed with "**t_**") in preparation for committing them to the persistent store.

We will showcase operations on HllSet nodes using a subset of data from the Enron Emails demo application, as discussed in our previous post [11].

The following is an excerpt from the graph we will be utilizing, as viewed in the Neo4J browser.



Node properties

row

<elementId>	4:931c3fc5-766d-4a34-80f5-bcf1c5346aca:149
<id>	149
card	18
d_sha1	22f6e1165872c25006130c246766df38d2c8e156
dataset	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0... Show all
labels	["row"]
props	{"Date": "1980-01-01 00:00:00","From": "phillip.allen@enron.com","To": "stephen.harrington@enron.com","commit_id": "6f6957f3-66e1-4a56-b186-0112b1d475... Show all
sha1	1c7335328c96de135cf28a6d9cf5d5e419646692

node_union

This function executes a union operation on a collection of nodes sourced from the "nodes" table, as detailed below:

```python
function node_union(db::DB, nodes::Vector; p::Int64=10)
      # Calculate sha1 ID for the new node that will represent
      # the result of union operation
      sha1s = []
      hll = SetCore.HllSet{p}()
      for node in nodes
          push!(sha1s, node.sha1)
          restored = SetCore.HllSet{p}()
          restored = SetCore.restore(restored, JSON3.read(node.dataset,
Vector{UInt64}))
          hll = SetCore.union!(hll, restored)
      end
```

```
        sha1 = Util.sha1_union(string.(sha1s))
        # Create all edges
        for _sha1 in sha1s
            new_edge = Edge(_sha1, sha1, "union", Config())
            replace!(db, new_edge)
        end
        # Create new node that will represent union
        d_sha1 = SetCore.id(hll)
        card = SetCore.count(hll)
        dataset = SetCore.dump(hll)
        props = Config()
        new_node = Node(sha1, ["union"], d_sha1, card, dataset, props)
        replace!(db, new_node)
        # Add sha1 of new node to the list sha1s
        push!(sha1s, sha1)
        return sha1s
end
```

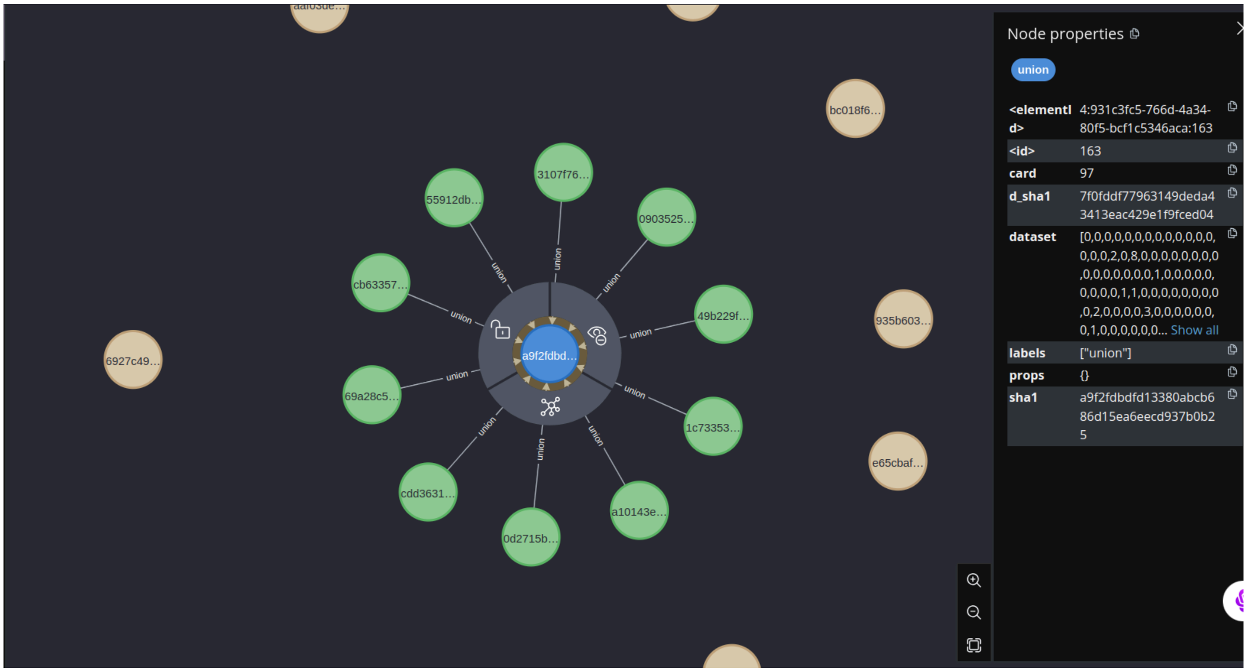
Below is the code that demonstrates the usage of this operation:

Python

```
# In this demonstration, we will be conducting a union operation on
the
# nodes that correspond to emails sent by 'phillip.allen@enron.com'.

query = raw"SELECT * FROM nodes WHERE json_extract(props, '$.From') =
'phillip.allen@enron.com'"

# Getting references
#
refs_rows = LisaMeta.select_nodes_by_query(db_meta, query, -1)

# Selecting all nodes from "nodes" table using list of sha1 ID (refs_row)
# as a reference
row_nodes = Vector()
LisaNeo4j.select_nodes(db_meta.sqlitedb, refs_rows, row_nodes)

# Performing union operation on the list of nodes (row_nodes)
refs = Graph.node_union(db_meta, row_nodes)
println("Union node added: ", refs)
```

```
# Updating Neo4J Graph DB with new nodes and edges
LisaNeo4j.add_neo4j_nodes_by_refs(db_meta.sqlitedb, Set(refs), url, headers)
LisaNeo4j.add_neo4j_edges_by_refs(db_meta.sqlitedb, Set(refs), url, headers)
```

Changes observed in the graph after the execution of this operation.



Node properties

union

<elementId>	4:931c3fc5-766d-4a34-80f5-bcf1c5346aca:163
<id>	163
card	97
d_sha1	7f0fddf77963149deda43413eac429e1f9fced04
dataset	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,3,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0... Show all
labels	["union"]
props	{}
sha1	a9f2fdbdfd13380abcb686d15ea6eecd937b0b25

Intersection based node operation

The operations in question share a structural similarity; therefore, we will illustrate their application using `**node_intersect**` as an example.

Below are the function signatures for the operations being implemented:

node_intersect:

```python
function node_intersect(db::DB, x::DataFrameRow, y::DataFrameRow; p::Int64=10)
```

node_comp:

```Python
function node_comp(db::DB, x::DataFrameRow, y::DataFrameRow; p::Int64=10)
```

node_xor:

```Python
function node_xor(db::DB, x::DataFrameRow, y::DataFrameRow; p::Int64=10)
```

node_diff:

```Python
function node_diff(db::DB, x::DataFrameRow, y::DataFrameRow; p::Int64=10)
```

node_intersect source code

This is the source code for the function that executes the `**node_intersect**` operation:

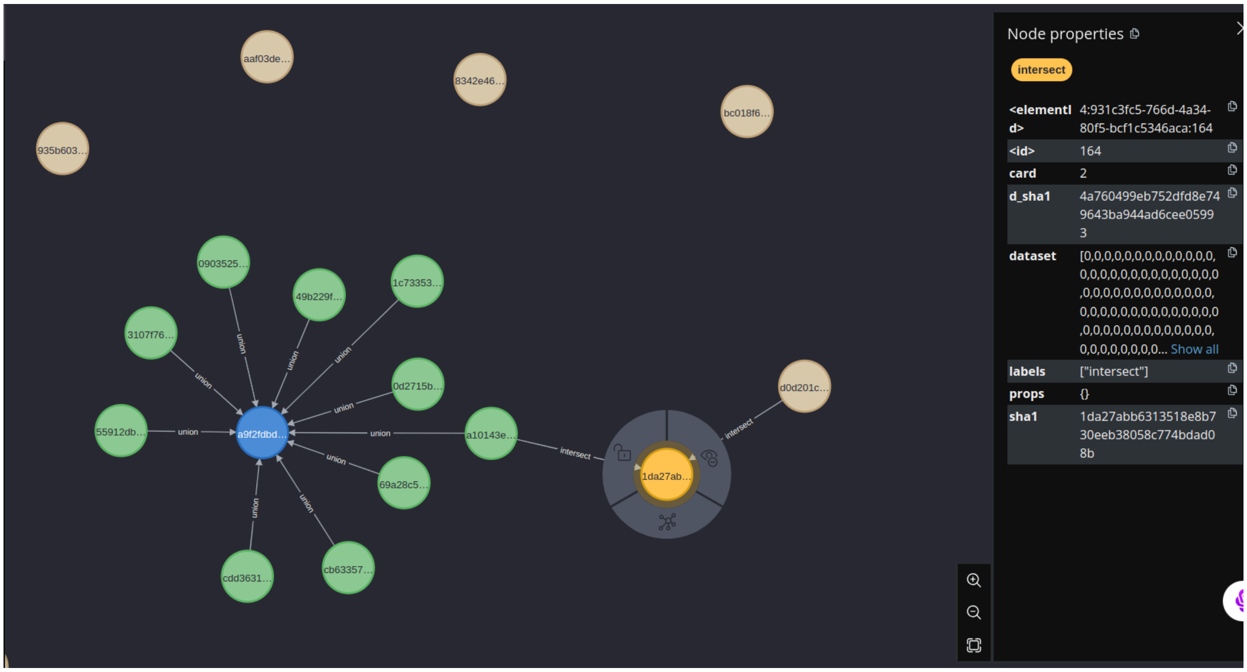
```Python
function node_intersect(db::DB, x::DataFrameRow, y::DataFrameRow; p::Int64=10)
    # Creating list of sha1 ID for operands
    sha1s = []
    push!(sha1s, x.sha1)
    push!(sha1s, y.sha1)
    # Restoring HllSets extracted from the nodes table
    hll = SetCore.HllSet{p}()
    restored_x = SetCore.HllSet{p}()
    restored_x = SetCore.restore(restored_x, JSON3.read(x.dataset,
Vector{UInt64}))
    restored_y = SetCore.HllSet{p}()
    restored_y = SetCore.restore(restored_y, JSON3.read(y.dataset,
Vector{UInt64}))
    # Obtaining the instance of intersection
    hll = SetCore.intersect(restored_x, restored_y)
    sha1 = Util.sha1_intersect(string.(sha1s))
    # Create all edges
    for _sha1 in sha1s
            new_edge = Edge(_sha1, sha1, "intersect", Config())
            replace!(db, new_edge)
    end
    # Creating a new node for the result of intersection
    d_sha1 = SetCore.id(hll)
    card = SetCore.count(hll)
    dataset = SetCore.dump(hll)
    props = Config()
    new_node = Node(sha1, ["intersect"], d_sha1, card, dataset, props)
    replace!(db, new_node)
    # Adding sha1 of new node to the list of sha1s
    push!(sha1s, sha1)
    return sha1s
end
```

Below is the source code to execute this operation:

Python

```
refs = Set(["d0d201c405adf0df65759c16de98f8ef4795d737",
"a10143e19099cd2679132f63bf59afd3e275bddf"])

row_nodes = Vector()
LisaNeo4j.select_nodes(db_meta.sqlitedb, refs, row_nodes)
println("Row nodes: ", row_nodes)

refs = Graph.node_intersect(db_meta, row_nodes[1], row_nodes[2])

LisaNeo4j.add_neo4j_nodes_by_refs(db_meta.sqlitedb, Set(refs), url, headers)
LisaNeo4j.add_neo4j_edges_by_refs(db_meta.sqlitedb, Set(refs), url, headers)
```

Here is the updated graph displayed in the Neo4J browser.



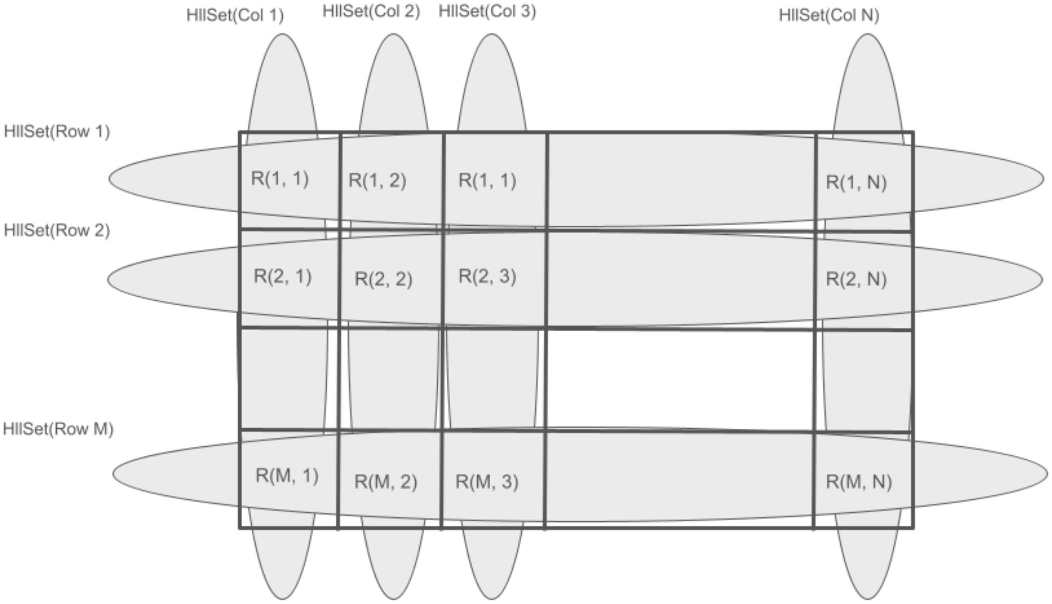
Node properties

intersect

<elementId>	4:931c3fc5-766d-4a34-80f5-bcf1c5346aca:164
<id>	164
card	2
d_sha1	4a760499eb752dfd8e749643ba944ad6cee05993
dataset	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0... Show all
labels	["intersect"]
props	{}
sha1	1da27abb6313518e8b730eeb38058c774bdad08b

projection

In our previous discussions, specifically in references [2] and [10], we delved into the structure of tabular data in relation to HllSets. When considering CSV files, the correlation seems quite intuitive: the columns in the CSV file correspond directly to columns in the HllSet presentations, and similarly, the rows in the CSV file align with rows in the HllSet structure.



We aim to extend this concept further by defining a tabular HllSet structure that can be applied to any two collections of HllSets. For the sake of organization, we will arbitrarily designate one collection as the 'column collection' and the other as the 'row collection'.

Every cell in the image depicted above represents an intersection of an HllSet that corresponds to a row and an HllSet that corresponds to a column. This implies that we can obtain the contents of each cell as an HllSet.

projection_1_1

We will begin by examining the case where one HllSet is projected onto another, which fundamentally represents an intersection. Therefore, we define **projection_1_1 as the node_intersection**.

projection_1_n

Here is the source code for the operation:

JavaScript

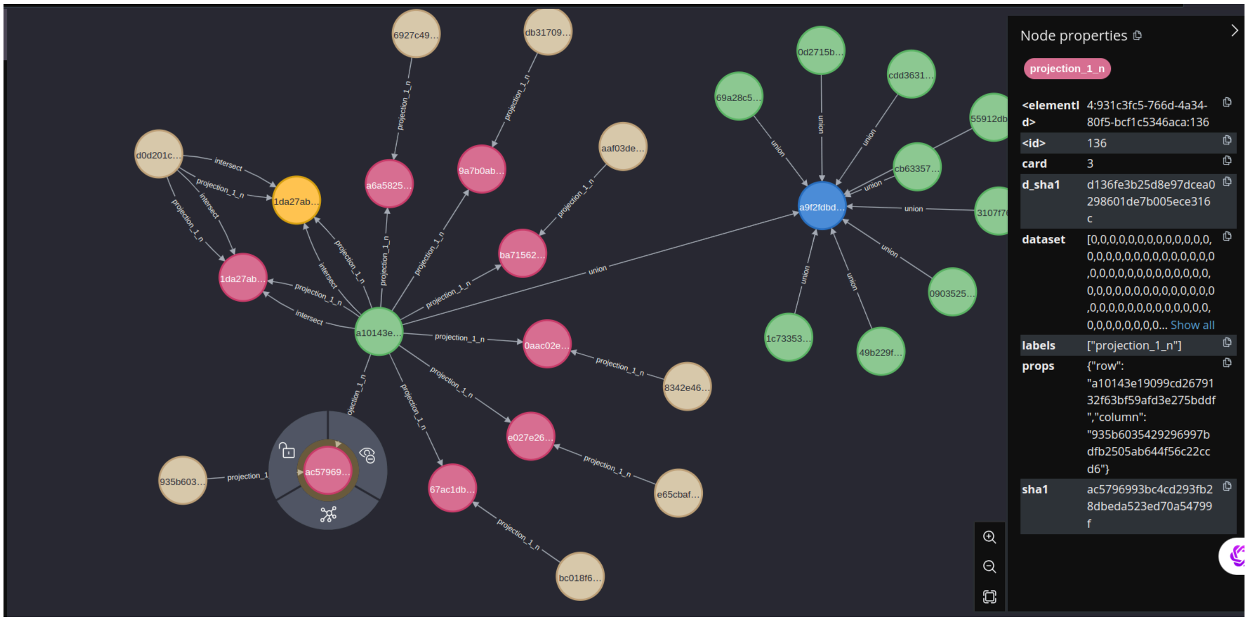
```
function projection_1_n(db::DB, row::DataFrameRow, cols::Vector;
label::String="projection_1_n", p::Int64=10)
    sha1s = []
    for col in cols
        sha1s = union(sha1s, node_intersect(db, row, col; label=label, p=p))
    end
    return sha1s
end
```

Here's an illustration of executing the operation projection_1_n:

JavaScript

```
row_ref = Set(["a10143e19099cd2679132f63bf59afd3e275bddf"])
col_refs = LisaMeta.select_sha1_by_label(db_meta, "column", "nodes", -1)
row_nodes = Vector()
LisaNeo4j.select_nodes(db_meta.sqlitedb, row_ref, row_nodes)

col_nodes = Vector()
LisaNeo4j.select_nodes(db_meta.sqlitedb, col_refs, col_nodes)

refs = Graph.projection_1_n(db_meta, row_nodes[1], col_nodes)

LisaNeo4j.add_neo4j_nodes_by_refs(db_meta.sqlitedb, Set(refs), url, headers)
LisaNeo4j.add_neo4j_edges_by_refs(db_meta.sqlitedb, Set(refs), url, headers)
```

Below is the graph that was generated in the Neo4J browser after executing this code:

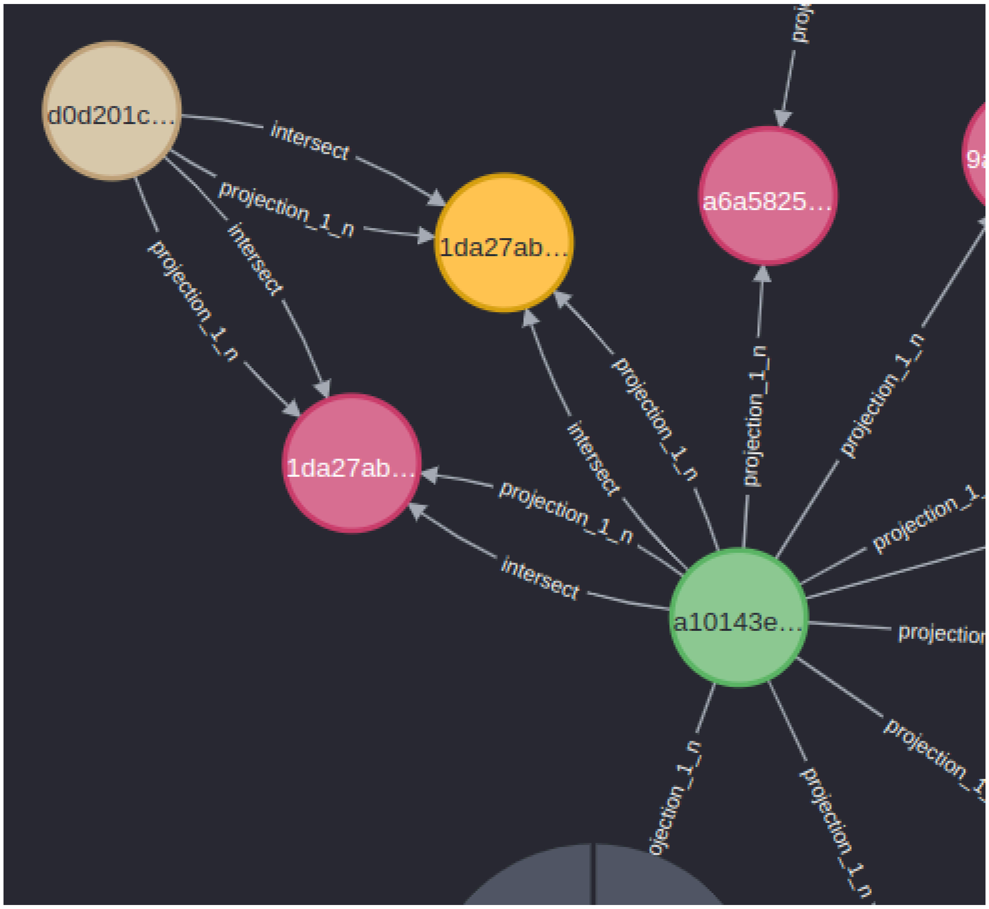


Node properties

projection_1_n

<elementId>	4:931c3fc5-766d-4a34-80f5-bcf1c5346aca:136
<id>	136
card	3
d_sha1	d136fe3b25d8e97dcea0298601de7b005ece316c
dataset	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0... Show all
labels	["projection_1_n"]
props	{"row": "a10143e19099cd2679132f63bf59afd3e275bddf","column": "935b6035429296997bdfb2505ab644f56c22ccd6"}
sha1	ac5796993bc4cd293fb28dbeda523ed70a54799f

Let's delve deeper into the most fascinating segment of this graph.



In this fragment, we observe two intermediate nodes sharing the same sha1 ID between the row-node (a10143e…) and the column-node (d0d201c…). This occurrence is a result of merging the outcomes of two operations (node_intersect and projection_1_n) on these respective nodes. Both operations generated a new node with the identical sha1 ID (1da27ab…) but with distinct labels - intersect and projection_1_n. This demonstrates a valuable feature of Neo4J, enabling us to prevent overlap in results generated by different operations.

projection_m_n

This version of the operation is a batch update of its predecessor. It enables the sequential execution of projection_1_n on m row-nodes. Below is the source code for this operation:

Python

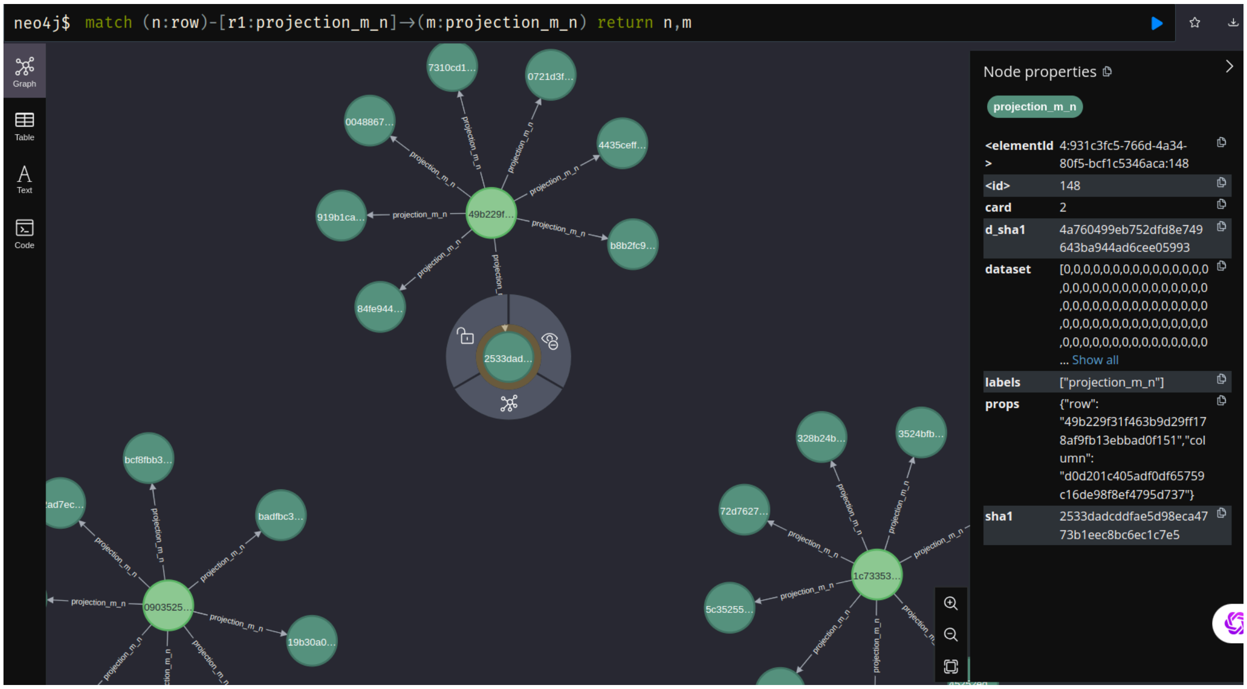
```
function projection_m_n(db::DB, rows::Vector, cols::Vector;
label::String="projection_m_n", p::Int64=10)
    sha1s = []
    for row in rows
        sha1s = union(sha1s, projection_1_n(db, row, cols))
    end
    return sha1s
end
```

Below is the code we are currently using to execute this operation:

Python

```
row_ref = Set(["1c7335328c96de135cf28a6d9cf5d5e419646692",
"49b229f31f463b9d29ff178af9fb13ebbad0f151",
"0903525a12d3a22b1538b7a6ecb9e220de17a5cc"])

row_nodes = Vector()
LisaNeo4j.select_nodes(db_meta.sqlitedb, row_ref, row_nodes)


refs = Graph.projection_m_n(db_meta, row_nodes, col_nodes)

LisaNeo4j.add_neo4j_nodes_by_refs(db_meta.sqlitedb, Set(refs), url, headers)
LisaNeo4j.add_neo4j_edges_by_refs(db_meta.sqlitedb, Set(refs), url, headers)
```

Here is the output displayed in the Neo4J browser:



```
neo4j$ match (n:row)-[r1:projection_m_n]→(m:projection_m_n) return n,m
```

Node properties

projection_m_n

<elementId>	4:931c3fc5-766d-4a34-80f5-bcf1c5346aca:148
<id>	148
card	2
d_sha1	4a760499eb752dfd8e749643ba944ad6cee05993
dataset	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ... Show all
labels	["projection_m_n"]
props	{"row": "49b229f31f463b9d29ff178af9fb13ebbad0f151","column": "d0d201c405adf0df65759c16de98f8ef4795d737"}
sha1	2533dadcddfae5d98eca4773b1eec8bc6ec1c7e5

HllSets Interpolation

By utilizing the HllSet operation, we are able to construct a series of intermediate HllSets between two designated HllSets, referred to as the source and the target. We can establish a gradation, which is the number of intermediate sets, to determine the number of transformation steps required to transition from the source HllSet to the target HllSet.

Summary

The document discusses the concept of HllSet Relational Algebra, which explores the relationship between relational algebra and HllSets, a data structure based on the HyperLogLog algorithm.

It covers the fundamental properties of HllSets, their mapping into a Graph DB, the identity problem in Computer Science, and the calculation of SHA-1 hashes for HllSets.

The text also delves into operations on HllSet nodes such as union, intersection, complement, and XOR, as well as the projection of HllSets designated as rows onto HllSets that are designated as columns.

Additionally, it provides code snippets and examples for executing these operations and updating a Graph DB accordingly. The text concludes with references to further reading material on the topic.

References

1. https://www.linkedin.com/posts/alex-mylnikov-5b037620_hyperloglog-based-approximation-for-very-activity-7191569868381380608-CocQ?utm_source=share&utm_medium=member_desktop
2. https://www.linkedin.com/posts/alex-mylnikov-5b037620_hllset-analytics-activity-7191854234538061825-z_ep?utm_source=share&utm_medium=member_desktop
3. https://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf
4. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/40671.pdf
5. https://github.com/alexmy21/lisa_meta/blob/main/lisa.ipynb
6. https://www.linkedin.com/posts/alex-mylnikov-5b037620_hllset-metadata-store-activity-7191836031090872320-83We?utm_source=share&utm_medium=member_desktop
7. https://en.wikipedia.org/wiki/Relational_algebra
8. https://dl.acm.org/doi/10.1145/358396.358400
9. https://en.wikipedia.org/wiki/Content-addressable_storage
10. https://medium.com/@alexmy_29874/introduction-to-general-theory-of-statistics-e20c0d17954a
11. https://www.linkedin.com/posts/alex-mylnikov-5b037620_demo-application-enron-email-analysis-with-activity-7195832040548614145-5Ot5?utm_source=share&utm_medium=member_desktop