HllSet Analytics

HllSet Analytics encompasses a range of topics focused on the application of set theory concepts to HllSets within the realm of metadata.

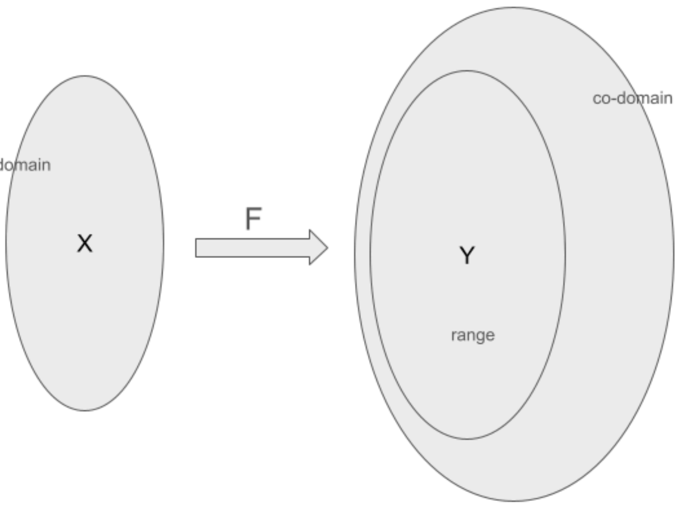
Before delving into this article, we highly recommend referring to the "HllSet Metadata Store" article in the previous post. It will provide valuable context and enhance your understanding of the topic discussed here.

Domains and codomains

In mathematics, the concept of a domain and co-domain is fundamental when discussing functions. Suppose we have a function:

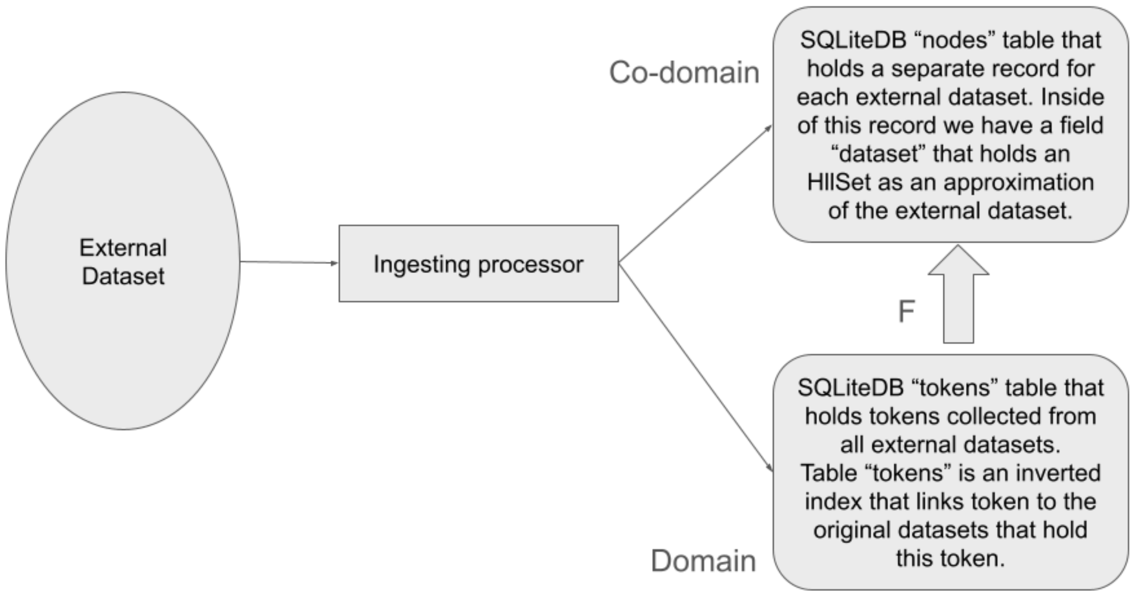
$$F\colon X \rightarrow Y$$

The domain of a function encompasses all possible inputs that the function can accept. Meanwhile, the co-domain represents the set of all potential outputs. Within this framework, X signifies a subset of elements drawn from the domain, and Y denotes a corresponding range of values within the co-domain that are associated with the elements from X.



In the HllSet metadata framework (HllSet-meta), the **domain** encompasses the various formats of elements found in external datasets. Ultimately, everything in computers is reduced to binary code. The distinction between text and images lies in the interpretation of these bits.

Within the HllSet-meta framework, this interpretation and conversion process is facilitated through the use of specialized processors designed to handle different data types. Currently, we have limited support for processing csv files, with a focus on extracting textual data (strings) from these files.


External Dataset
Ingesting processor
Co-domain
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.
F
Domain
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.

In this diagram, each **token** from the inverted index is transformed into a **hash**—specifically, an unsigned 64-bit integer. This **hash** then acts as a **token ID** within the index of **tokens**.

For the **nodes** table, each token ID is incorporated into a **HllSet** designated for a specific external dataset. The HllSet is a specialized structure, not merely a collection of elements. It consists of a predetermined number of rows, referred to as **bins**, with each **bin** being a 64-bit BitVector.

The total number of rows in this structure is a power of **2**, denoted as $\mathbf{2^p}$, where we currently utilize $\mathbf{p = 10}$. This configuration keeps the HllSet's size relatively compact, approximately 8KB, while still providing a reliable estimate of cardinality. Importantly, the size of the HllSet remains constant and is independent of the size of the original dataset. This means any dataset, regardless of its size, can be represented within an HllSet no larger than 8KB. For those instances where a more precise estimate of cardinality is required, the precision can be increased by using a larger value of **p**.

The algorithm for assigning token IDs to bins is straightforward:
1. First, we calculate the bin number by using the first **p** bits of the token ID's hash value. For instance, if **p** equals 10, the range of possible bin numbers is from 0 to 1024.
2. Next, we determine the count of consecutive zeros at the end of the token's hash value in binary form. This count cannot exceed **64 - p**.

Consequently, the total number of potential combinations for all HllSets is given by $2^{(64 - p)^{2^p}}$, which represents an extremely large figure. Yet, this number, vast as it may be, is still smaller than the total number of potential combinations of all tokens in the tokens index. It's also clear that different hashes from the tokens table may result in identical **(bin, zeros)** pairs when represented in the HllSet.

The function **F** serves as a "many-to-one" mapping from the tokens to their representations in the HllSet. This implies that tracing a **(bin, zeros)** pair from an HllSet back to a specific token in the tokens table could lead to multiple tokens. Without access to the original dataset, determining the correct token among several options becomes challenging, if not impossible. Additionally, it's conceivable for more than one traced token to have originated from the original dataset, due to the low, yet nonzero, risk of hash value collisions.

In conclusion, once an original dataset is transformed into an HllSet, pinpointing the exact source of the HllSet's representation becomes difficult.

In the **lisa_analytics.ipynb** [1], we will explore methods to lessen the ambiguity involved in tracing tokens back from their HllSet representations.

The accuracy of converting datasets into HllSets relies on two crucial factors:

- the precision parameter **p**, which determines the level of detail in the conversion by controlling the bin count, and
- the selection of the **hash function**, particularly how its output varies based on the seed values used during initialization. By adjusting these seed values, we have the ability to impact the resulting hash values.

Here is an outline of the algorithm:

1. Initially, we process the dataset using a standard hash function:
   $$\mathbf{F(std): X(std) \rightarrow Y(std)}$$
2. Next, we track the original tokens by applying reverse processing:
   $$\mathbf{G(std): Y(std) \rightarrow X(std)}$$
3. Subsequently, we repeat the dataset processing using the same hash function but with different seed values:
   $$\mathbf{F(seed): X(seed) \rightarrow Y(seed)}$$
4. We then reverse the results from the modified hash function:
   $$\mathbf{G(seed): Y(seed) \rightarrow X(seed)}$$

It is evident that the standard and seeded results may not align, particularly when dealing with large datasets:

$$\mathbf{X(seed) \neq X(std)}$$

However, it is expected that tokens from the original dataset should be present in both sets.

Does this method eliminate uncertainty? It is possible, but not guaranteed. We anticipate that the intersection of the two results would provide a more accurate representation:

$$\mathbf{X = X(seed) \cap X(std)}$$

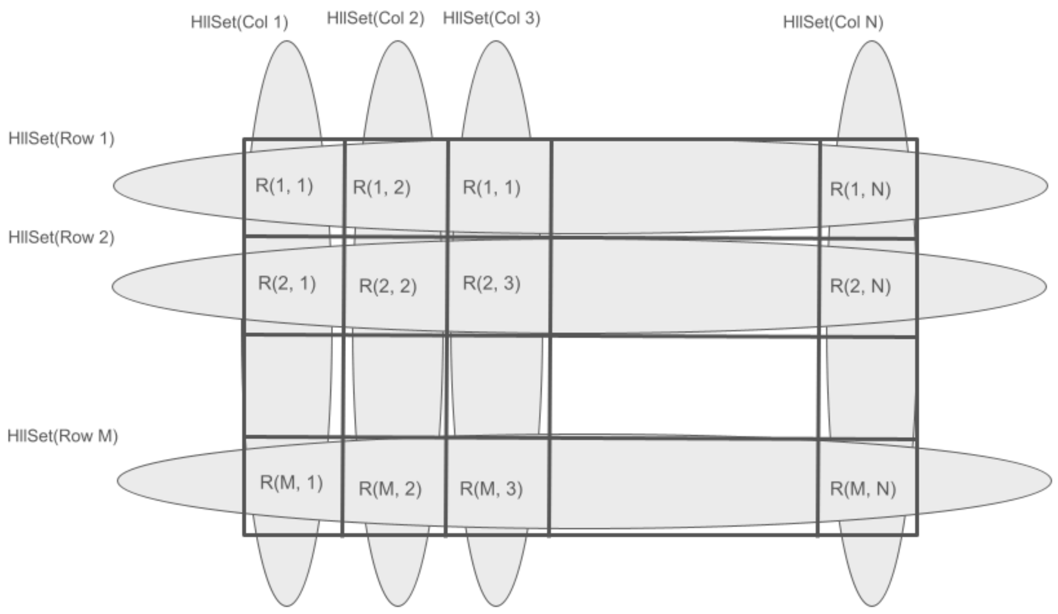
To observe how this technique performs, we will apply it to some sample data in the **lisa_analytics.ipynb** file.

Applying HllSets for Tabular data structures

Formally, tabular data sets can be described as a subset of the Cartesian product of two sets:

$$R(S_1, S_2) \subseteq S_1 \times S_2$$

However, in the scenario of HllSet, the direct access to set elements is not available. Therefore, an alternative method must be employed in order to construct and utilize table structures.



The illustration above demonstrates how binary relationships can be created between HllSets. For instance, when dealing with CSV files, the process would involve the following:

- **S_1** - a collection of HllSets representing lines from the CSV file
- **S_2** - a collection of HllSets representing columns CSV file.

Each cell in the table view represents the intersection of an HllSet row and an HllSet column, resulting in a new HllSet.

All elements within an HllSet are part of a common SSI dictionary (tokens) and are assigned the same identifiers used during the construction of the HllSet. This allows for easy tracing of the contents back to their corresponding tokens in the dictionary, enabling the restoration of cell contents.

When working with HllSets, the following considerations should be taken into account:

- The contents of matrix cells consist of an unordered collection of tokens from the original CSV cell. To determine the order of these tokens, reference to the original CSV file is necessary.
- In many instances, only a subset of rows are utilized, which must be kept in mind when working with the data.

The correction process is typically straightforward:

- To obtain the actual HllSet for a column, the union of the HllSets of the cells within that column must be used.
- The same approach applies to working with HllSets.

An example of using metadata for searching in SSI

We'll look at two examples of using metadata:

1. Recovering tabular data from CSV files;
2. Search for CSV files and columns that match words from the search query.

Recovering tabular data

Python
```python
'''
First we need to generate metadata for the columns and rows of the files we are
going to work with.
During processing, we will add metadata to two main tables of the Graph
Database:
- nodes;
- tokens.
In the nodes table we will place row and column data for each file. Each row
and column will receive a unique SHA1 identifier.
All new words from the files will be entered into the tokens table, indicating
the SHA1 identifiers of the rows or columns of the files from which these words
were selected.
This will establish a relationship between the tokens (words) and the elements
of the files they were in.

db - Graph Database;
row.name() is a function that retrieves the full filename from the row object
that represents the file.
'''
```

```
for row in eachrow(file_list)
    Store.ingest_csv_by_row(db, row.name())
end

for row in eachrow(file_list)
    Store.ingest_csv_by_column(db, row.name())
end
```

The Graph Database (Store object) has specialized methods for extracting data matrices from the intersections of rows and columns of CSV files.

```Python
"""
    Here we are going to extract the row and column nodes from the csv file.
    The resulting matrix will show the number of elements at the intersection
of the row and column nodes.
"""
matrix = Store.get_card_matrix(db, source_id)
for row in each row(matrix)
    println(row)
end
```

Output after executing this program:

```Python
[2.0, 2.0, 2.0, 1.0, 2.0, 4.0, 3.0, 3.0]
[2.0, 2.0, 1.0, 1.0, 2.0, 4.0, 5.0, 4.0]
[2.0, 3.0, 2.0, 1.0, 1.0, 4.0, 4.0, 4.0]
[2.0, 2.0, 2.0, 1.0, 1.0, 4.0, 4.0, 3.0]
[2.0, 2.0, 2.0, 1.0, 2.0, 4.0, 3.0, 3.0]
[2.0, 3.0, 2.0, 1.0, 1.0, 4.0, 3.0, 3.0]
[2.0, 2.0, 2.0, 1.0, 2.0, 4.0, 3.0, 3.0]
[2.0, 3.0, 2.0, 1.0, 2.0, 4.0, 3.0, 3.0]
[2.0, 2.0, 1.0, 1.0, 2.0, 4.0, 3.0, 3.0]
. . .
```

As we see, many matrix cells have several elements.

```
Python
matrix = Store.get_value_matrix(db, source_id)
for row in eachrow(matrix)
    println(row)
end
```

And this is the output from this program (only the first few lines of output are shown):

```
Python
["[\"Serious\"]", "[\"Pedestrian\"]", "[\"Male\"]", "[]", "[\"Friday\"]",
"[\"Kensington\",\"Chelsea\",\"and\"]", "[\"Motorcycle\",\"over\",\"and\"]",
"[\"Not\",\"Pedestrian\",\"pedestrian\"]"]
["[\"Serious\"]", "[\"Pedestrian\"]", "[\"Female\"]", "[]", "[\"Wednesday\"]",
"[\"Kensington\",\"Chelsea\",\"and\"]",
"[\"car\",\"Taxi/Private\",\"and\",\"hire\"]",
"[\"crossing\",\"Pedestrian\",\"Crossing\",\"ped.\",\"facility\"]"]
["[\"Serious\"]", "[\"rider\",\"Driver\"]", "[\"Male\"]", "[]", "[\"Monday\"]",
"[\"Kensington\",\"Chelsea\",\"and\"]",
"[\"cycle\",\"Motor\",\"and\",\"under\"]",
"[\"crossing\",\"carriageway\",\"elsewhere\"]"]
["[\"Serious\"]", "[\"Pedestrian\"]", "[\"Male\"]", "[]", "[\"Monday\"]",
"[\"Kensington\",\"Chelsea\",\"and\"]",
"[\"cycle\",\"Motor\",\"and\",\"under\"]",
"[\"Not\",\"Pedestrian\",\"pedestrian\"]"]
["[\"Serious\"]", "[\"Pedestrian\"]", "[\"Male\"]", "[]", "[\"Sunday\"]",
"[\"Kensington\",\"Chelsea\",\"and\"]", "[\"Car\",\"and\"]",
"[\"Not\",\"Pedestrian\",\"pedestrian\"]"]
["[\"Serious\"]", "[\"rider\",\"Driver\"]", "[\"Male\"]", "[]",
"[\"Tuesday\"]", "[\"Kensington\",\"Chelsea\",\"and\"]",
"[\"Motorcycle\",\"over\",\"and\"]", "[\"Not\",\"pedestrian\"]"]
. . .
```

Finding nodes in a graph containing words from a query

In this scenario, we utilized Neo4J Graph as the server for our Graph Database. To interact with Neo4J, we employed a custom-built interface called LisaNeo4J [2]. This interface offers essential functionalities for graph operations, such as data retrieval.

The following snippet illustrates a program using simplified Julia code to outline the fundamental procedures for handling a request.

Python
```
# Find nodes and edges (SHA1 identifiers) that contain "sex", "taxi" and "day"
rows = LisaNeo4j.search_by_tokens(db.sqlitedb, "sex", "taxi", "day")
# Fetch all edge references
edges = Vector()
edges_refs = LisaNeo4j.select_edges(db.sqlitedb, rows, edges)

# Fetch all links to nodes
nodes = Vector()
LisaNeo4j.select_nodes(db.sqlitedb, refs, nodes)

# Building a presentation in Neo4J
#1. Let's start with nodes
for node in nodes
    labels = replace(string(node.labels), ";" => "")
    query = LisaNeo4j.add_neo4j_node(labels, node)
    data = LisaNeo4j.request(url, headers, query)
end
# 2. After this, connect the nodes with edges
for edge in edges
    query = LisaNeo4j.add_neo4j_edge(edge)
    data = LisaNeo4j.request(url, headers, query)
end
```

Search results enable us to conduct fundamental analytics. In this instance, we are aiming to assess the connections between CSV files that meet our specified query criteria. We compute two key metrics: Jaccard distance and Cosine similarity.

Python
```
# Defining a Cypher request
query = LisaNeo4j.cypher("MATCH (n:csv_file) RETURN n.labels, n.sha1, n.d_sha1,
n.dataset, n.props LIMIT 20")

hlls = LisaNeo4j.collect_hll_sets(query, hlls)
# Define relationships between related csv files
for (k, in) in hlls
    for (d1, v1) in hlls
        if k != k1
            jaccard = SetCore.jaccard(v.hll_set, v1.hll_set) # jaccard
comparison
```

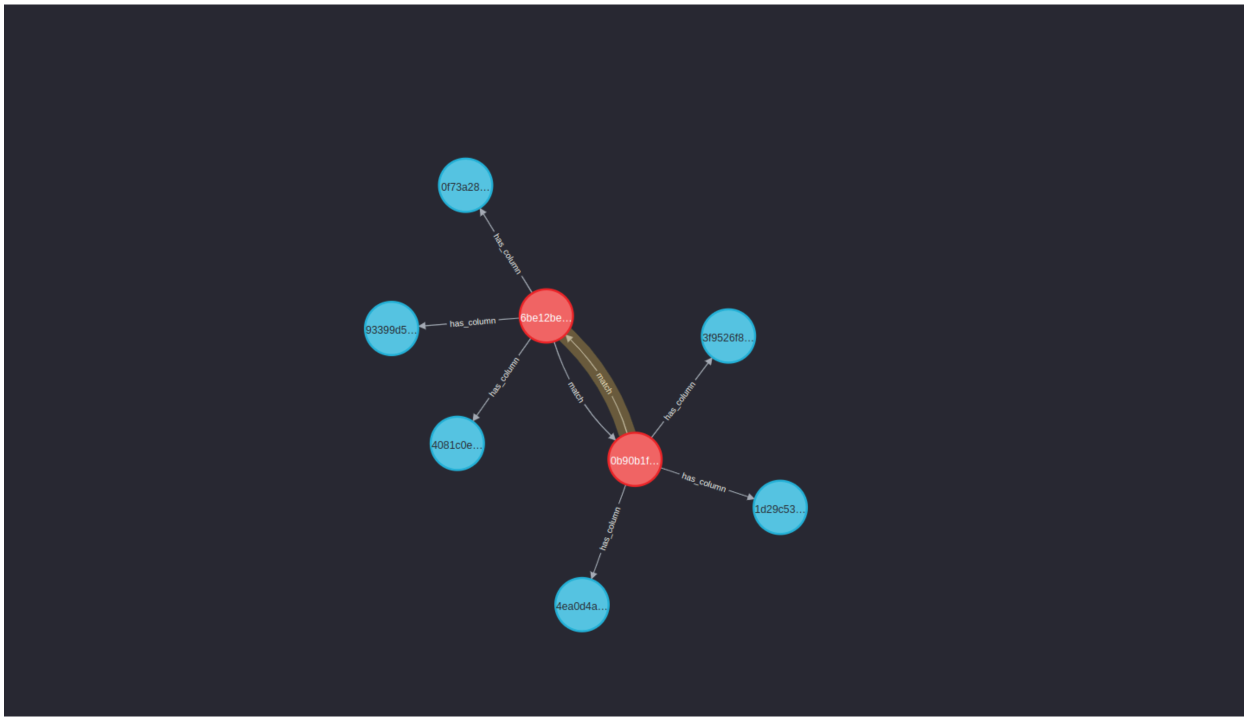
```
      println("jaccard: ", jaccard)
            cosine = SetCore.cosine(v.hll_set, v1.hll_set)  # cosine comparison
      println("cosine: ", cosine)
         end
    end
end
```

And this is the conclusion from this program.

Unset

```
jaccard: 78
cosine: 87.0
jaccard: 78
cosine: 87.0
```

In our example, we utilized 2 files, resulting in one relationship being measured by two different methods. It is important to highlight that we obtained two relationships for each metric due to the directed nature of our graph. In directed graphs, symmetric connections are depicted by two edges: one from the first node to the second, and another from the second to the first. Additionally, there is a notable discrepancy in the values of these two metrics.



Summary

HllSet Analytics introduces a novel approach to handling and analyzing metadata through the application of set theory concepts, specifically designed for HllSets. The foundation of this method lies in the transformation of datasets into highly compressed, efficient representations that facilitate various types of data analysis, including cardinality estimation and dataset comparison.

Key Concepts and Processes:

1. **Domains and Codomains**: The article explains these fundamental mathematical concepts as they apply to functions, with a focus on their relevance to the HllSet metadata framework. This framework interprets and processes different data types, particularly text from csv files, through specialized processors.

2. **HllSet Structure**: It is detailed how elements from datasets are represented within an HllSet, a structure consisting of a fixed number of bins (rows), each being a 64-bit BitVector. This structure allows for a compact representation of datasets, regardless of their original size, with a standard HllSet being about 8KB.

3. **Token IDs and Bins**: The process of assigning token IDs to bins involves calculating the bin number from the token ID's hash value and determining the count of consecutive zeros at the end of the token's hash in binary form. This method allows for a many-to-one mapping from tokens to their HllSet representations, introducing challenges in tracing back to the original tokens without the original dataset.

4. **Algorithm for Converting Datasets into HllSets**: The article outlines a two-step hashing process, using standard and modified seed values, to manage and potentially reduce the uncertainty in identifying original tokens from HllSet representations.

5. **Application to Tabular Data**: HllSets are applied to represent and analyze tabular data structures, with methods described for constructing and utilizing table structures through the intersection of HllSet rows and columns.

6. **Metadata Generation and Recovery**: The process involves generating metadata for rows and columns of CSV files and adding this metadata to a Graph Database. This enables the recovery of tabular data from CSV files by establishing relationships between tokens (words) and the file elements they originate from.

7. **Graph Database and Neo4J Integration**: The article showcases how to use a Graph Database, specifically Neo4J, to find nodes and edges containing specific tokens, and how to visualize these connections. This integration facilitates the analysis of CSV files that match search queries.

8. **Analytical Metrics**: The use of Jaccard distance and Cosine similarity metrics is demonstrated for assessing relationships between CSV files, highlighting the potential discrepancies in values due to the directed nature of the graph.

Conclusion:

HllSet Analytics presents an innovative framework for metadata analysis, offering a compact, efficient way to represent and analyze large datasets. Through the use of specialized algorithms and structures, it enables the handling of diverse data types and the exploration of relationships within and between datasets. The integration with graph databases and analytical metrics further enhances its utility for data analysis applications.

References

1. https://github.com/alexmy21/lisa_meta/blob/main/lisa_analytics.ipynb
2. https://github.com/alexmy21/lisa_meta/blob/main/lisa_neo4j.ipynb