Heliyon 7 (2021) e08035

Contents lists available at ScienceDirect

Heliyon

journal homepage: www.cell.com/heliyon

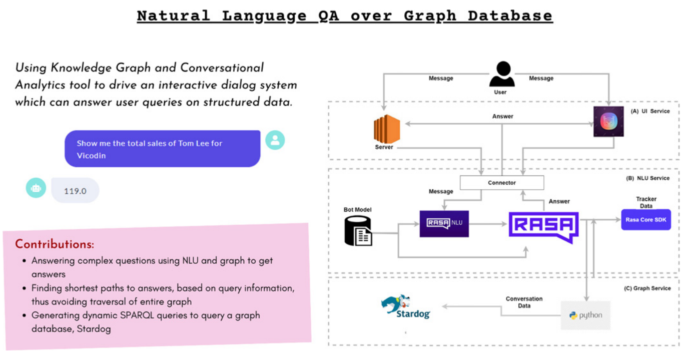
Research article

Graph NLU enabled question answering system

Sandeep Varma, Shivam Shivam [*], Snigdha Biswas [*], Pritam Saha, Khushi Jalan

ZS Associates, Tower 3, World Trade Centre, Kharadi, Pune, Maharashtra 411014, India

GRAPHICAL ABSTRACT



Natural Language QA over Graph Database

Using Knowledge Graph and Conversational Analytics tool to drive an interactive dialog system which can answer user queries on structured data.

Show me the total sales of Tom Lee for Vicodin

119.0

Contributions:

- Answering complex questions using NLU and graph to get answers
- Finding shortest paths to answers, based on query information, thus avoiding traversal of entire graph
- Generating dynamic SPARQL queries to query a graph database, Stardog

ARTICLE INFO

Keywords:
Conversational analytics
Graph traversal
Knowledge graph
Natural language query
Question answering
Structured data
Tabular data

ABSTRACT

With a huge amount of information being stored as structured data, there is an increasing need for retrieving exact answers to questions from tables. Answering natural language questions on structured data usually involves semantic parsing of query to a machine understandable format which is then used to retrieve information from the database. Training semantic parsers for domain specific tasks is a tedious job and does not guarantee accurate results. In this paper, we used conversational analytics tool to create the user interface and to get the required entities and intents from the query thus avoiding the traditional semantic parsing approach. We then make use of Knowledge Graph for querying in structured data domain. Knowledge graphs can be easily leveraged for question answering systems, to use them as the database. We extract appropriate answers for different types of queries which have been illustrated in the Results section.

1. Introduction

A lot of data and information today is stored in the form of structured data and the need of information extraction from these data has been constant. Effectively and precisely locating information is vital for many applications, hence, careful search and analysis of the information in tabular data is important. Question Answering (QA) systems return answers for natural language queries given by users. QA in structured data usually involves semantic parsing to transform the natural language query to some machine understandable logical form which is then used to retrieve answers from tables ([1, 2]).

Natural language queries can be simple questions involving single entity and predicates around it, or complex questions involving multiple entities and predicates among them. While simple queries are easier to

* Corresponding authors.
E-mail addresses: shivam.shivam@zs.com (S. Shivam), snigdhasawsib2000@gmail.com (S. Biswas).

https://doi.org/10.1016/j.heliyon.2021.e08035

handle and have been dealt using semantic parsers earlier, complex queries arise the need to find answers by combining multiple aspects which are relatively difficult to handle.

Knowledge graphs (KG) in the recent years have been used to store large amounts of data in industrial and research areas. It is an efficient way of storing unstructured data due to its predicate-argument structure. It is also being used to store structured data as well. Compared to the rigidity of a relational database structure, a KG can maintain multiple points of view simultaneously. KGs can handle querying in huge datasets better than relational databases, and it is also easier to add new data sources to it.

We have tried to solve in this paper some of the problems that are associated with answering complex natural language queries.

- Methods using semantic parsers convert the natural language queries to some logical form which, can be used to query tables. But semantic parsers require large training which gets difficult for domain-specific data. The logical forms of the query are error-prone, which can lead to reduced accuracies. And although some models have achieved great results for simple natural language queries, there haven't been many attempts in cases of complex natural language queries. We, therefore, try to eliminate the need to use semantic parsers by using Natural Language Understanding (NLU) tools instead. We don't just try to answer complex queries with multiple entities and predicates but also try to extend it to queries involving filter and aggregation.
- We answer queries from structured data (tables), but we make use of KG instead of relational databases. As datasets increase along with the relationships between them, querying relational databases get complex. KGs, however, excel at querying huge data. It is easier to define new relations within data in KG and can model complex data relationships. Another advantage of KG, which is used in industries extensively, is the ability to visualize data. It can combine multiple dimensions of each data node. And given its use to store unstructured data as well makes it possible to combine both structured and unstructured data on a single platform.

Using KG to drive an interactive dialog system which can answer user queries from tabular data has been attempted in this paper. Various graph database supporting knowledge curation, querying and modifying make the entire process lot simpler and efficient. One such graph database, Stardog [3] has been used here. The tabular data is stored as a graph model in Stardog. Instead of semantic parsers, we have used conversation analytics tool to extract intent and entities from the query. The extracted entities and intent are used to query the graph to get to the specific answers. Our approach can fetch answers across multiple tables and can answer complex questions. The absence of semantic parsers helps minimize erroneous logical forms of the queries and also saves us from the supervised training requiring large computational cost. Training is inexpensive and can be easily trained for any domain specific data. The rest of the paper is divided into the following sections: Section 3 gives a brief description of the data used, Section 4 describes the general idea of the overall architecture, followed by detailed descriptions. Section 5 demonstrates the results that have been obtained from the approach followed by conclusion and future works. The main contributions of our work can be summarized as follows:

- We have tried to answer complex questions which involves filtering and aggregation queries as well, using NLU and shortest path traversal on graph to get answers.
- We have created an algorithm for subgraph creation in runtime to optimize the response time for each query.
- We have developed a method to find shortest paths to answers, based on query information, thus avoiding traversal of entire graph to get the answer.
- We have designed a method to generate dynamic SPARQL queries which is used to query a graph database, Stardog.

2. Related works

Many models using semantic parsing perform well for simple questions. Berant et al. have focused on the problem of semantic parsing natural language utterances into logical forms that can be executed to produce denotations [4]. Yih et al. have also proposed a similar approach to learn a semantic parser that maps a natural language question to a logical form query that can be executed against a knowledge base to retrieve the answers [5].

Semantic parsers however, are supervised and need large initial training which may not be feasible for domain specific data. Such data are difficult to annotate. The recent model TAPAS ([6] presents a weakly supervised technique for QA over tables without having to generate any intermediate logical form of the queries. Fine-tuning TAPAS for domain specific tabular data can be expensive. TAPAS currently also cannot be used for larger tabular data containing more than 500 rows. However, the idea of eliminating the need to get intermediate logical structures of query saves us a lot of incorrect answers which can result due to generation of flawed intermediate forms.

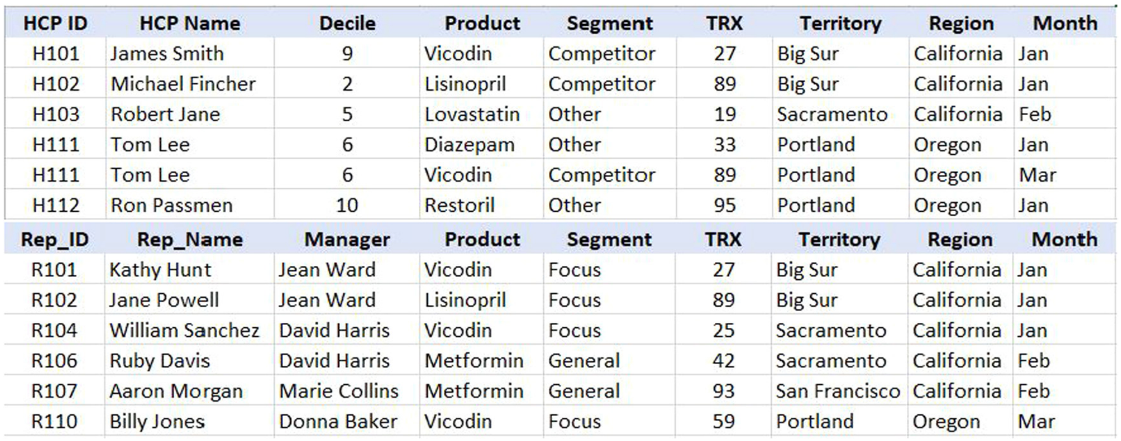
Huang et al. make use of knowledge graph embeddings for question answering system for simple queries. Each predicate/entity is represented as a low-dimensional vector, so as to preserve the relation information in the Knowledge Graph (KG). However, the different ways a predicate can be expressed and ambiguity of entity names make it challenging [7].

Abujabal et al. have presented an automated template-based QA system over knowledge graphs where they tried to answer complex questions also. But the need of dependency parsers limits their results where they fail to identify templates for questions due to incorrect dependency parse trees and POS tag annotations [8].

Zheng et al. have shown that knowledge graphs have proved to be an efficient way in the recent years with the huge amounts of structured and linked data. Query languages like SPARQL, which although are a powerful way of querying such Knowledge Base (KB), are difficult for casual users who are not familiar with the syntax as well as back-end data structures of such KB. Natural language queries thus provide a more promising approach ([9]). Zhu et al. [10] have discussed several works on QA over KB. Many focus on translating the natural language query to SPARQL queries, whereas others make use of manually designed rules. In contrast to earlier works, they focused on finding the most appropriate path. Their work is however limited to non-aggregation type of queries.

Lu et al. [11] have proposed a model 'QUEST' which attempts at answering complex queries from multiple unstructured documents with the use of Quasi Knowledge Graph. QUEST dynamically retrieves documents related to the query and generates subject-predicate-object triples. These triples are used to create graph containing nodes and edge weights. The shortest paths connecting the query entities are hypothesized to contain the answers.

We have used a concept similar to 'QUEST' using graph traversal to find answers from tabular/structured data. The method is applied over Resource Development Framework (RDF) which is queried using SPARQL queries. We have drawn inspiration from [10] to generate SPARQL templates based on our graph traversal method, which does not limit the templates to a few question types and also eliminates parsers and manually designed rules. Eliminating them helps avoiding errors as well as excessive training. Along with complex questions involving multiple entities and relationships, we also aimed to fetch answers for aggregation and filter queries.



HCP ID	HCP Name	Decile	Product	Segment	TRX	Territory	Region	Month
H101	James Smith	9	Vicodin	Competitor	27	Big Sur	California	Jan
H102	Michael Fincher	2	Lisinopril	Competitor	89	Big Sur	California	Jan
H103	Robert Jane	5	Lovastatin	Other	19	Sacramento	California	Feb
H111	Tom Lee	6	Diazepam	Other	33	Portland	Oregon	Jan
H111	Tom Lee	6	Vicodin	Competitor	89	Portland	Oregon	Mar
H112	Ron Passmen	10	Restoril	Other	95	Portland	Oregon	Jan

Rep_ID	Rep_Name	Manager	Product	Segment	TRX	Territory	Region	Month
R101	Kathy Hunt	Jean Ward	Vicodin	Focus	27	Big Sur	California	Jan
R102	Jane Powell	Jean Ward	Lisinopril	Focus	89	Big Sur	California	Jan
R104	William Sanchez	David Harris	Vicodin	Focus	25	Sacramento	California	Jan
R106	Ruby Davis	David Harris	Metformin	General	42	Sacramento	California	Feb
R107	Aaron Morgan	Marie Collins	Metformin	General	93	San Francisco	California	Feb
R110	Billy Jones	Donna Baker	Vicodin	Focus	59	Portland	Oregon	Mar

Fig. 1. A small subset of the data set.

3. Dataset and example queries

The dataset that has been used comprises of real-time tabular data with details of various health care professionals (HCPs), customers, products, sales across various territories and regions and in various months of the year for a company. The information has been stored across multiple tables, which are, interconnected with each other. The data is HCPcentric and describes at a very granular level, the products an HCP are prescribing, the territory in which he works, the amount of sales he's making with every prescription, and the exact segment in which the HCP falls. This data has been described using relations in the graph, where HCP is the central node and all the related data emerge out from that node. All the HCPs are associated with a unique identifier which helps in distinguishing between them. We have used **geographical data** which shows the alignments of sales representatives across various territories and the sales they are making which helps them in getting the required compensation. Just like the HCPs, the sales reps have also been assigned unique identifiers which help in distinguishing between them. The data shows things like exact territory and regions where he works, the manager under whom he's working, and the amount of sales made for which he needs to be compensated for at different times. There are separate **customer** tables for these reps, contains various details like customer id, address, corporate parent id, etc. and tables for **product details** containing the market names of product, market id, etc. There are many other tables like timetables, sales. A snippet of two of the tables is shown in Fig. 1.

In the experiment done in this paper, the following types of queries have been tried to extract answers for:

- **Queries with no filter or aggregation:** These are queries with a single or multiple entities. Taking examples from Fig. 1- *"What are the products prescribed by Tom Lee?"*
- **Queries with filter:** These queries require some kind of filtering. For example, *"What are the products prescribed by Tom Lee except Vicodin?"*
- **Queries with aggregation:** Some kind of aggregation operation like sum or count is performed for these queries. If from Fig. 1, we ask *"What is the total sales (trx) under the manager Jean Ward?"*, the answer should be $27+89=116$.
- **Queries with filter and aggregation:** Both filter and aggregation operation are required for these types of queries. For example, in the query *"Which product has the highest sales in California except Lisinopril?"*, we need to perform aggregation operation to find rank of all products in terms of sales and then also have to filter out *Lisinopril*.
- **Queries with spelling errors:** As the name suggests, these queries involve entities with incorrect spellings. *"Who are the HCPs in Califrnia?"* where the spelling of *California* is incorrect.
- **Queries with missing entities:** These are incomplete queries like *"Find about Vicodin."*, where the chatbot cannot identify what to find and asks the user to check the question.

4. Methodology

4.1. Overview of architecture

We need to answer questions from multiple tables, so the first step of our approach is to combine all the tabular data into triples. A triple is a set of three entities that represents semantic data in the form of subject–predicate–object. Once we have the triples, it acts as the knowledge base from which information can be retrieved. The next step is the architecture which interacts with the user to get the answers of the questions. This architecture majorly consists of two parts:

1. The conversational system, fetching query from the user in the correct format and extracting the entities and intents
2. Traversing the knowledge base to get to the required answer.

The first part of the architecture is taken care of by RASA, an open source natural language understanding tool in chatbots ([12]). It performs entity extraction and intent classification (answer type) on the queries received. The fallback mechanism takes place if the query is not formed correctly. It also takes care of incomplete queries where it asks the user for more details to complete the query. The flow diagram for this part is shown in Fig. 2.

The second part traverses the graph, searching for the desired answer based on the query entities. The method finds out the destination nodes in the graph, as per the query, and uses a shortest path method to reach from the source nodes (entities) to destination nodes. The next sections describe each step of the implementation in details.

4.2. Generating triples for graph modeling

We have generated triples from all the tabular data in the format of *Primary_Entity-Class-Entity*. In order to retrieve answers from multiple tables, there must be some relation between the tables. The primary keys are normally taken as the feature for joining multiple tables. We have made use of the primary key of all the tables for triple creation process. The primary key values are the *Primary_Entity* of the triples. For each row of the tables, all other column values have been linked with the primary key with the class or relation between the two being the column name. Thus, the primary keys act as the *Primary_Entity*, all other column values act as the Entity; and the Class joining the two entities have been taken from the column names of the corresponding Entity.

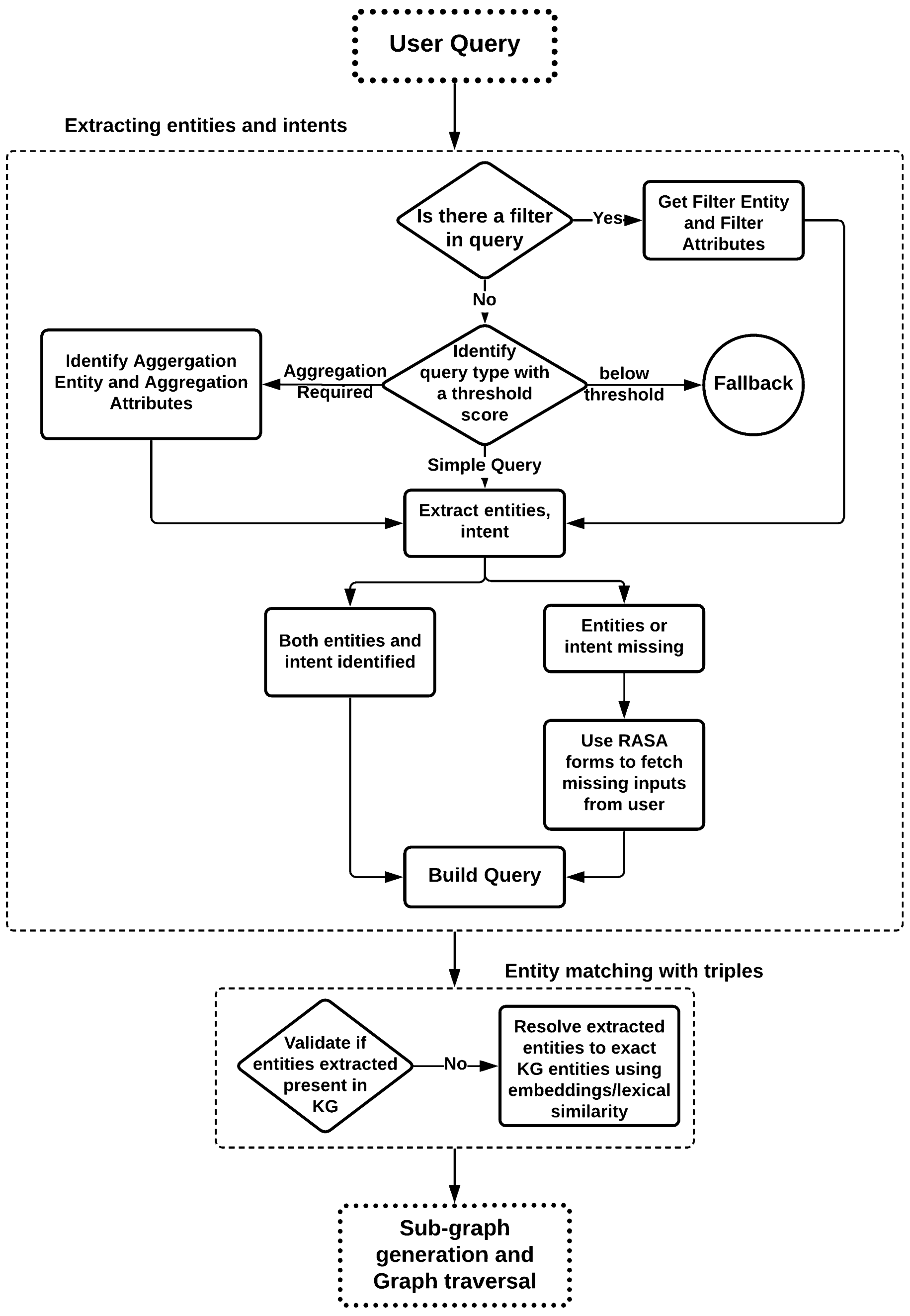

User Query
Extracting entities and intents
Is there a filter in query
Yes
Get Filter Entity and Filter Attributes
No
Identify query type with a threshold score
Aggregation Required
Identify Aggregation Entity and Aggregation Attributes
below threshold
Fallback
Simple Query
Extract entities, intent
Both entities and intent identified
Entities or intent missing
Use RASA forms to fetch missing inputs from user
Build Query
Entity matching with triples
Validate if entities extracted present in KG
No
Resolve extracted entities to exact KG entities using embeddings/lexical similarity
Sub-graph generation and Graph traversal

Fig. 2. Workflow of query processing.

For cases where tabular data did not contain any primary key, indexing the table has been done. The indexed values in the format *'filename_row no.'* have been taken as the primary keys for each row. This additional column has been taken as the *Primary_Entity* and the rest of the triple has been created in the same way as described above. As an example, from Fig. 1. *'hcpdata_1 – product – Vicodin'*; *'hcpdata_1 – HCP name – James Smith'* are two different triples that is generated.

One single triples file is thus created from all the tables which acts as our database for fetching answers to questions. The graph model has been created from these triples based on the question asked.

4.3. Extracting entities and classes

Conversational analytics provides the interface to ask queries and show the results to the users. On getting the query, the first task is to extract entities and classes from the queries. We define entities and classes as follows:

Entity: Any specific node in the graph (any specific cell in the tabular data).

Class: The type of the entity node (column name of the entity cell in tabular data).

The graph algorithm needs the entities in the questions and their class types as the input along with the class type of the answer needed. For example, in the query- *In which region is the territory San Francisco located?*; the entity is *San Francisco* with its class as *territory*. The answer required is of the class type *region*. The graph will give *California* as the answer to this question, which has the class type *region*.

We give as input to the graph algorithm the following values:

Query_Entities: A list of all the entities in the question is provided in this. From the above example, query entity is *San Francisco*.

Query_Ent_Classes: A list of all the classes of the entities in the question. *Territory* belongs to this as per the given example.

Answer_Classes: This is the class type of the required answers. We want to know the region where *San Francisco* is located. So, *Answer_Class* is *Region*.

For queries requiring some filtering of answers like: *'What are the territories in California except San Francisco?'*, we need to filter San Francisco from the answer and should be passed as an input to the graph algorithm.

For an aggregation question of aggregation type *'max'*: *What is the largest territory in California in terms of sales?* We want to find the territory whose sales value is maximum. Both these information needs to be passed to the graph algorithm and hence needs to be extracted from the query.

Given below are few more terminologies that has been used for filtering and aggregation types of questions:

Filter_Flag: This is kept as 1 if the query type needs any filtering otherwise remains 0. If 1, two inputs- Filter_Entity and Filter_Class are passed to the graph algorithm.

Filter_Entity: This is the node that needs to be filtered from the query. In the above example, Filter Entity is *San Francisco*.

Filter_Class: This is the class type of the Filter_Entity.

Territory is the Filter Class in the above question.

Aggregate_Flag: This is kept as 1 if the query type needs any aggregation function otherwise remains 0.

Aggregation_Type: The aggregation function that needs to be performed. Currently in our algorithm, there are the following types of aggregation functions- **Count, Sum, Min, Max, Rank, Threshold, Similarity, Popularity**

Aggregation_Class: This is the class according to which aggregation needs to be performed. In the above example, we want largest territory in terms of sales value. So, sales will be the Aggregation_Class.

Aggregation_Entity: If aggregation result of some particular entity is asked, the entity is named as Aggregation_Entity. If we want to find *the rank of Portland in terms of sales, Portland* is taken as the Aggregation_Entity.

Synonyms of various Answer_Classes have been stored in lookup tables so that queries having intents different from those in triples can also be detected. For example- if instead of 'HCP name', the query contains 'prescriber', lookup tables will identify it as a synonym of 'HCP name' and accordingly provide input to the graph.

We have used Conversational Analytics Tool, RASA for the extraction process. RASA has two main components-

- **RASA NLU:** A library for Natural Language Understanding (NLU) that classifies intent, and extracts entities of the query, and helps the chatbot to take action accordingly.
- **RASA CORE:** A chatbot framework that takes the structured input from the NLU and predicts the next action using a probabilistic model.

Model Description

In RASA, the model consists of different parts and each part is defined sequentially in a pipeline. Each query passes through every part in the pipeline, and the intent is classified, and entities are extracted by the RASA NLU. Based on intent classified the RASA CORE takes the necessary action. The model pipeline that has been used consists of the following-

- A custom made unit for correcting spelling errors before entities are extracted by RASA NLU.
- Tokenization and featurization to compute representations for the examples of training data.
- Synonym mapper.
- Featurizer to create features for intent classification.
- Named Entity Recognizer for entity extraction.

Algorithm 1 Entity Matching Algorithm.

```
1: $similarity\_score \leftarrow []$
2: for $n$ in $nodes$ do
3:     $similarity\_score_i \leftarrow levenshtein\_distance(entity, n)$
4: end for
5: if $max(similarity\_score) > threshold$ then
6:     $entity \leftarrow nodes_i$
7: end if
```

- DIET Classifier - a multi-transformer architecture for intent classification and entity recognition ([13]).

The model is currently tested and used for production.

Every type of query needs only a few sentences to train and hence training is easier. Once all the above-mentioned values are obtained, graph is used for getting the answers.

4.4. Entity matching with triples

The user asking the question does not know the exact entities or classes present in our database. For example: the user may mistakenly type California as Califrnia, As the triples will be converted to graph, we need the exact entities and intent from the query as is present in the triples. Thus, comes the need for string matching. We search the triples to get most similar nodes using Algorithm 1. The similarity score of each entity with all nodes of graph is found. A threshold value is set for the similarity score and the node is selected only if the score is higher than the threshold.

4.5. Fetching answers from graph

We have all entities and classes extracted by RASA with us, and they are present in our database, which has been ensured by the entity matching with triples step. From here, two ways of implementation of fetching answers from graph have been tried. The basic concept behind both implementations (as will be discussed in the subsequent sections) is the same.

The first one is a simple implementation of the idea on Python's NetworkX package ([14]). As will be shown later, the method proves inefficient as the size of database keeps increasing. So, a modification of the implementation has been done, using a graph database, Stardog which is queried using SPARQL, an RDF query language.

4.5.1. Approach using NetworkX

NetworkX is a Python package for creating, manipulating, and studying complex networks' structure, dynamics, and functions. Data structures for graphs, digraphs, and multigraphs are supported by it. It also includes some standard graph algorithms. It allows for visualizations of graphs in 2D and 3D.

In this approach the generated triples are simply stored in a NetworkX graph, which is traversed to get to the answers. All the entities present in the triples form the nodes in the graph and the Class of the entities act as the edges between them as discussed in Section 5. The graph is undirected and unweighted graph i.e., each edge of the graph has a unit weight. The graph traversal is divided into the following two parts.

Creation of sub-graph

The triples generated from all the tabular data can be huge, and hence creating the KnowledgeGraph with the complete triples and performing graph algorithms on it can be computationally expensive. Hence, we dynamically generate sub-graphs containing all the nodes related to the query entities. The sub-graph thus created is much smaller and is easier to handle. If the entities are Primary_Entity all nodes connected to it are taken for the generation of the sub-graph. If entities are of type Entity, we get all nodes connected to the Primary_Entity of that Entity (Algorithm 2).

Algorithm 2 Subgraph Generation Algorithm.

```
1: if type($entity$) = Primary_Entity then
2:     $Graph.nodes \leftarrow neighbors(entity)$
3: end if
4: if type($entity$) = Entity then
5:     if type(neighbors($entity$)) = Primary_Entity then
6:         $pr\_entity \leftarrow neighbors(entity)$
7:         $Graph.nodes \leftarrow neighbors(pr\_entity)$
8:     end if
9: end if
```

Algorithm 3 Algorithm for Shortest Path to Answer Nodes.

```
1: $cornerstones \leftarrow entities$
2: for $nodes$ in $Graph.nodes$ do
3:     if type($nodes$) = Primary_Entity then
4:         $primary\_keys \leftarrow nodes$
5:     end if
6: end for
7: for $cnode$ in $cornerstones$ do
8:     for $pnodes$ in $primary\_keys$ do
9:         $path_i, path\_length_i = ShortestPath(cnode, pnodes)$
10:    end for
11:    $Min\_path\_length = min(path\_length)$
12:    if $path\_length_i = Min\_path\_length$ then
13:        $primarykeys\_cnode \leftarrow path_i[-1]$
14:    end if
15:    $primarykeys\_cornerstone_i \leftarrow primarykeys\_cnode$
16: end for
17: $intersection\_primarykeys \leftarrow$ Intersection($primarykeys\_cornerstone$)
18: for $keys$ in $intersection\_primarykeys$ do
19:     if type(neighbors($keys$)) = Answer_Class then
20:         $destination\_nodes_i \leftarrow$ neighbors($keys$)
21:     end if
22: end for
```

Shortest path traversal of the subgraph

This method has been inspired from the technique used in 'QUEST' ([11]). We define the entity nodes of the queries as *cornerstones*. We find all the nodes of the graph having the answer class (intent) as it's description. We name them as the *destination nodes*.

The traversal can be majorly broken into two steps: Finding all primary key nodes connected to the entity nodes, followed by finding those destination nodes which are connected to the *primary key nodes*. From each *cornerstones*, we get one path to each of the primary key nodes. We select those primary key nodes whose path lengths are the shortest among the rest for all the *cornerstones*. Same is done for obtaining *destination nodes* from the primary keys. The above steps are done using Algorithm 3.

For simple queries like, *'What are the names of prescriber in California region'* having just one entity *'California'*, we just find the primary keys and destination nodes connected to that entity node (There can be multiple primary keys having shortest paths and hence multiple destination nodes as is expected from the answer). For queries having multiple entities, the shortest path to the primary keys is found from each individual cornerstone. The intersection of all the obtained primary keys gives us those primary keys which are common in all the shortest paths to the cornerstones. We then obtain the answers by traversing to the destination nodes. The final answer nodes can be single or multiple depending upon the query.

4.5.2. Approach using Stardog

Stardog is a semantic graph database. It also connects the data at the compute layer instead of the storage layer. Stardog allows one to explore and query knowledge graphs. Stardog can be used to run SPARQL queries on a knowledge graph. Stardog provides support for RDF data model. In this model, edges and nodes are defined in terms of subject-predicate-object triples where subjects and objects are nodes and the relationship is shown by the predicate. Schema definition can be customized using Stardog, as it provides support for RDF Schema (RDFS) and Web Ontology Language (OWL).

Algorithm 4 Shortest Path on Ontology Graph.

```
1: $cornerstones \leftarrow entities$
2: $destination \leftarrow Answer\_Class$
3: for $cnode$ in $cornerstones$ do
4:     $source_i \leftarrow type(cornerstones)$
5:     $path_i, path\_length_i = ShortestPathDFS(source_i, destination)$
6: end for
```

To integrate Stardog in our approach, we first need to load our triples into the Stardog database. And since Stardog uses SPARQL as a query language, we need to dynamically create SPARQL templates as per the question, which will also incorporate the shortest path concept between source and destination nodes. The steps under this are discussed in the next subsections.

Loading data into StarDog

As discussed in Section 5, the triples have been generated by connecting every column or attribute of the tables to the primary key column of each table. The ontology of the graph is defined similarly. The ontology does not contain the actual entities but the skeleton of the graph, that is, how the different classes of entities are connected with each other. Protégé [15] has been used for this purpose. After the creation of ontology, a named graph has been created in Stardog. All the data has been loaded to this named graph. The graph as expected is undirected and unweighted.

Shortest path traversal of ontology graph

Unlike in the method earlier described, where we generated a subgraph and calculated the shortest paths to get to the answers on that subgraph (which consisted of all the entities and the relationships), in this approach, we don't traverse the actual graph. Rather we generate an ontology graph in Python, which as said earlier, shows how the different classes of entities are connected to each other and do not contain the actual entities.

A part of the ontology graph is shown in Fig. 3. The nodes colored in *Orange* show the *Primary_Entity* nodes of the tables which are connected to all other entity classes of that table. The *Grey* colored nodes show the *Classes* of the *Entities* and how they are connected to the *Primary_Entity* type nodes. The ae nodes are the same data which have been described by HCP details in Section 3. We have the product table and customer tables as described in dataset (*cust_table_row_number and product_table_row_number). The geo table is connected with ae_geography_id, but since this is just a small part of the ontology, it is not shown here.*

As we know, we have from RASA the extracted values of *Query_Ent_Classes* and *Answer_Classes*. This means that we know the *Class* type of our *cornerstones/source nodes* and *destination nodes*. We find the shortest paths between these *Class* nodes of the *source* and *destination nodes* from the ontology graph.

Depth First Search (DFS) algorithm is used for this graph traversal, which has a time complexity of O(V + E), where V is the number of nodes/vertices and E is the number of edges (Algorithm 4). The ontology graph has fewer number of nodes and edges, as it just has the different class types and the connections and not the actual data. Hence the time required for this step is very less compared to finding shortest path on graph involving all nodes and edges.

We store the shortest path and the path length from this step which is used in the next step, to form the SPARQL queries dynamically. In case of multiple *entities*, shortest paths from all *classes* of the *entities* are found.

Dynamically generating SPARQL queries

Based on the length of the shortest path calculated in the earlier step, the SPARQL queries have been created to fetch actual shortest path results from the Stardog graph, which contains all the entities. The traversal has been done along the path calculated from the ontology graph, that is, the ontology based shortest paths specify the path taken to get to the actual results in the named graph.

We define the final SPARQL query as *Main* query. This *Main* query constitutes of several *Child* queries depending on the length of path. A *Child* query has been generated for each pair of nodes in the shortest

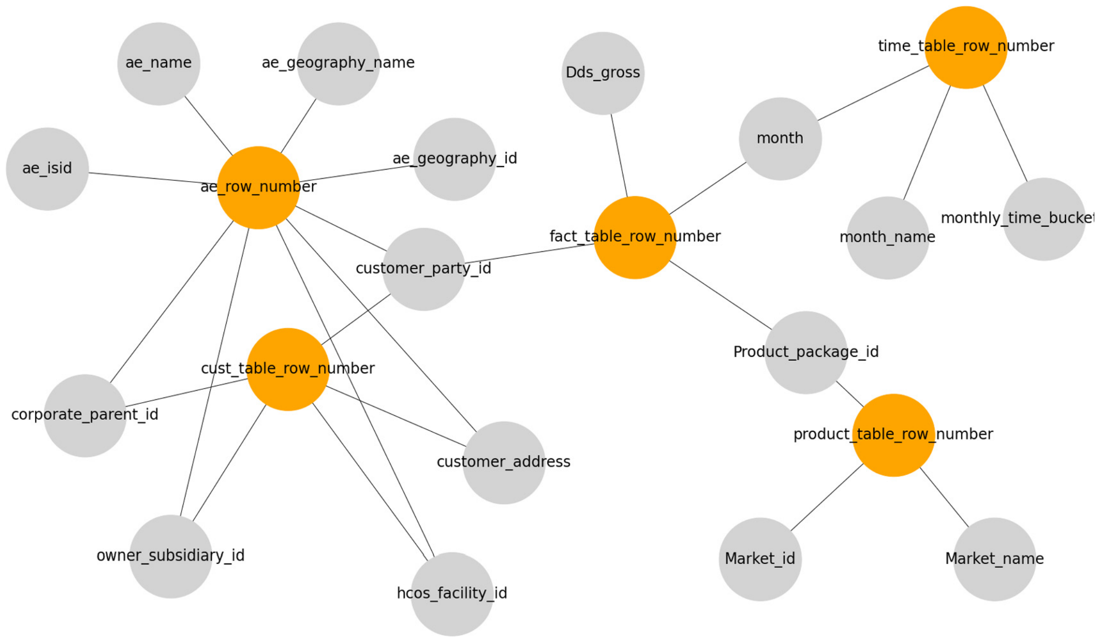


Fig. 3. Ontology graph in Python.

Algorithm 5 SPARQL query generation from shortest paths.

```
1: $N \leftarrow$ no._of_nodes($path_i$)
2: $Child\_query_0 \leftarrow$ SPARQL_template($entity$, $path_i[1]$)
3: for $k$ in $(1, N)$ do
4:     $Child\_query_k \leftarrow$ SPARQL_template($path_i[k]$, $path_i[k+1]$)
5: end for
6: $Main\_query \leftarrow \{Child\_query_0, Child\_query_1, ..., Child\_query_{N-1}\}$
```

paths, obtained from the aforementioned step. So, if the length of the shortest path consists of N nodes, including *source* and *destination* nodes, we have N-1 edges between them. So, we get N-1 *Child* queries for that *Main* query. We start with the first *Child* query which is the relation between the actual value of the *entity* and *Class* type of the next node as per the shortest path. This is continued for N-1 times to reach the final destination node. It is illustrated in Algorithm 5.

4.6. Additional query types

Apart from queries involving single or multiple entities, we have queries for filter and aggregation types as mentioned earlier. For filtering and aggregation type of queries, a few more steps need to be included after the earlier mentioned steps.

When we get the *Filter_Flag* as 1, along with all the steps mentioned in previous sections, we have also obtained the primary key nodes for the *Filter_Entity*. From the primary key nodes obtained from the cornerstones, we have then removed the primary keys obtained from the *Filter_Entity*. We have thus gotten the *destination* nodes from the now filtered *primary key* nodes.

In aggregation type queries the following functions have been implemented-

- **Sum**: This is used to give total values of the *destination* nodes. A sum operation is performed on the *destination* nodes obtained from the above method.
- **Count**: As the name suggests, it gives the total number of *destination* nodes obtained for a query.
- **Max**: This function can be used for two purposes. It can give the maximum value of the *destination* nodes obtained. This can take place if the destination nodes are numeric as in the query: *'What is the maximum sale in San Francisco?'* It can also answer queries like *'What is the largest territory in California in terms of sales?'* In this case the destination nodes have class type territory which is not numeric. However, it is mentioned in the question to get the largest territory in terms of sale. We first find all the *destination* nodes, i.e., the territories in California region. We then get the sales values of each of these territories and show the largest territory based on the maximum sales value calculated.
- **Min**: Min function has similar purpose like that of max for calculating minimum value. Queries can be of both the types: *'Give me the minimum sales of Tom Lee'* and *'Who is the prescriber with lowest sales?'*
- **Rank**: This is used for query types asking for ranks. Like max and min aggregations, we find the *destination* nodes from the *cornerstones* and rank them based on the class mentioned in the question. Then the rank of the entity asked is given as the result.
- **Threshold**: This deals with queries involving some threshold value like: *'Show me all the low decile prescribers'*, where the value of threshold decile may be provided by the user or a default threshold value may be set.
- **Similarity**: This is another type of query which is used to get nodes similar to the entity mentioned in the question. As an example query: *'Show prescribers similar to Tom Lee.'* returns those prescribers which have most neighboring nodes common with that of *Tom Lee's*.
- **Popularity**: This type is used for queries like *'Show me the most popular prescriber in a region.'* where we find the *Answer_Class* node (here, *prescriber* type node) which is most densely connected with other nodes.

For queries where the aggregation is to be simply performed on *Answer_Class* nodes, the steps remain the same with just the addition of performing the aggregation operation on the *destination* nodes. The queries where aggregation is to be performed based on some given *Aggregation_Class* require minor changes. The SPARQL query generation in this case has been modified to include both the *destination* nodes and the *Aggregation_Class* related to those destination nodes within the same *Main* query.

There may be queries where no entities are present like the one in the threshold function. In those type of queries, we have taken destina-

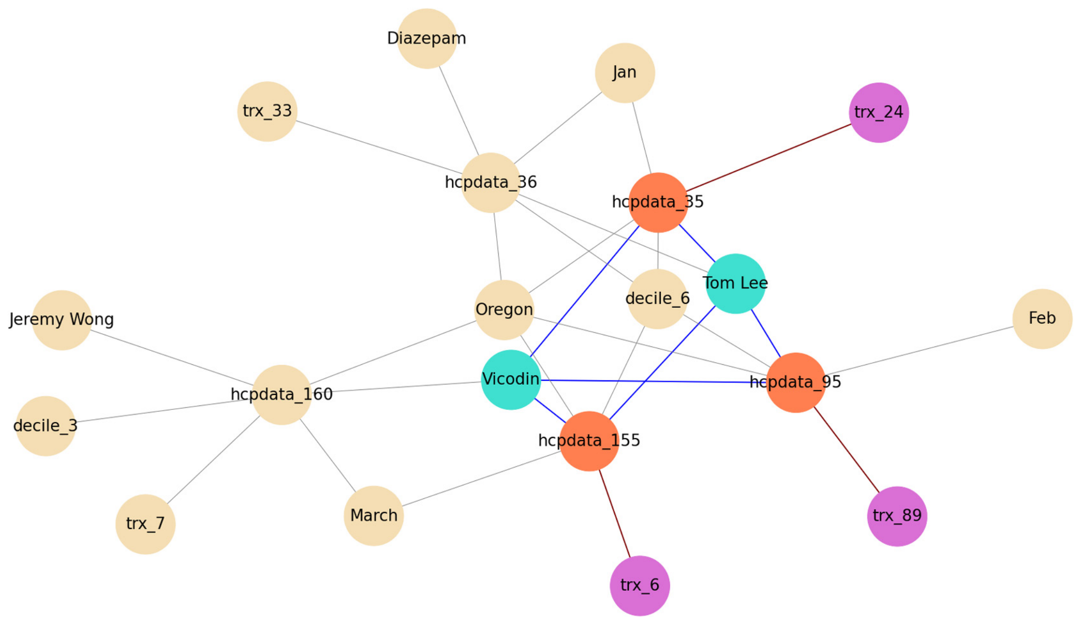


Fig. 4. Subgraph for the query *'How much Vicodin did Tom Lee sell?'*

tion nodes as the cornerstones and perform the necessary aggregation on them to get the desired destination/answer node.

4.7. A brief of complete methodology with example

We take here an example query to go over the complete approach. Let us have the following question:

Get all market names of products prescribed by Tom Lee.

Referring to Section 3, we have *Tom Lee* under the column *HCP Name*. We have all the products sold by her in the same table. But as explained earlier, there is a separate *Product* table, which contains details of the products. So, we need to fetch answers across two tables.

All the tables are already stored in triples format. We need to first use RASA to extract entities and intents from the query. The second step is using this information to traverse the graph.

For the first step of extracting entities and intents, RASA first checks if there is any filter in the query. In this case, no filter is present so it proceeds to aggregation checking. It gets no aggregation in this case as well. It then extracts the entities and intents. It returns the *Query_Entity* as *Tom Lee*, *Query_Ent_Class* as *HCP Name* and *Answer_Class* is *Market_name*.

After this step, the extracted entity (*Tom Lee*) is matched with the triples to check if the node is present in the database. In this case, we have the node *Tom Lee*, so it proceeds to the next step to fetch answers by traversing the graph.

The graph traversal (check Fig. 3) occurs along the shortest path which is

ae_name –> ae_row_number –> customer_party_id –>
fact_table_row_number –> Product_package_id –>
product_table_row_number –> Market_name

We get multiple *market names* of products, as *Tom Lee* prescribes multiple products.

More detailed explanation of graph traversal for *Aggregation* and *Filter* queries is discussed in the next section.

5. Results and discussions

We show below step wise results for few of the questions and explain the traceability of the solution through graph visualization. It will be followed by a discussion of the experimental results observed. Graph visualization for both the approaches used (shortest path on entire graph in 'NetworkX' and shortest path on ontology graph in 'Stardog') has been shown below.

5.1. Shortest path approach on entire graph

Aggregation Query:

How much Vicodin did Tom Lee sell?

This question is related to a physician's sales value for a particular product. This is an aggregation query with *Aggregation_Type- Sum* having two *Query_Entity- 'Tom Lee'* and *'Vicodin'* with *Answer_Class- 'sales'*. These values are extracted by RASA which then goes as input to the graph algorithm.

Fig. 4. shows a part of the subgraph for this question. The nodes colored in *Green* are the *cornerstones (Query_Entity)* which are connected to several primary key nodes. The primary key nodes colored in *Orange* are the intersection of the primary keys we obtain from the first step of graph traversal which are connected to both the entities (The paths colored in *Blue*). The second step is getting the sales values (mentioned as trx in the figure) from the common primary keys, shown by the *Brown* colored paths. The last step is performing aggregation operation *Sum* on these trx nodes colored in *Violet*.

Aggregation Query with Filter:

Show me the most prescribed drug in Portland except Vicodin?

Here we want to understand the maximum prescribed drugs in a particular region leaving one particular drug. The values extracted for this query will be *Aggregation_Type- Max* having *Query_Entity- 'Portland'*, *Answer_Attribute- 'product'* with *Filter_Entity* being *'Vicodin'*. The graph visualization for this question is given in Fig. 5.

After getting all *primary keys* connected to the *cornerstone Portland*, we eliminate those associated with *Vicodin* as can be seen by the primary key nodes marked in *Red*. The rest of the primary key nodes

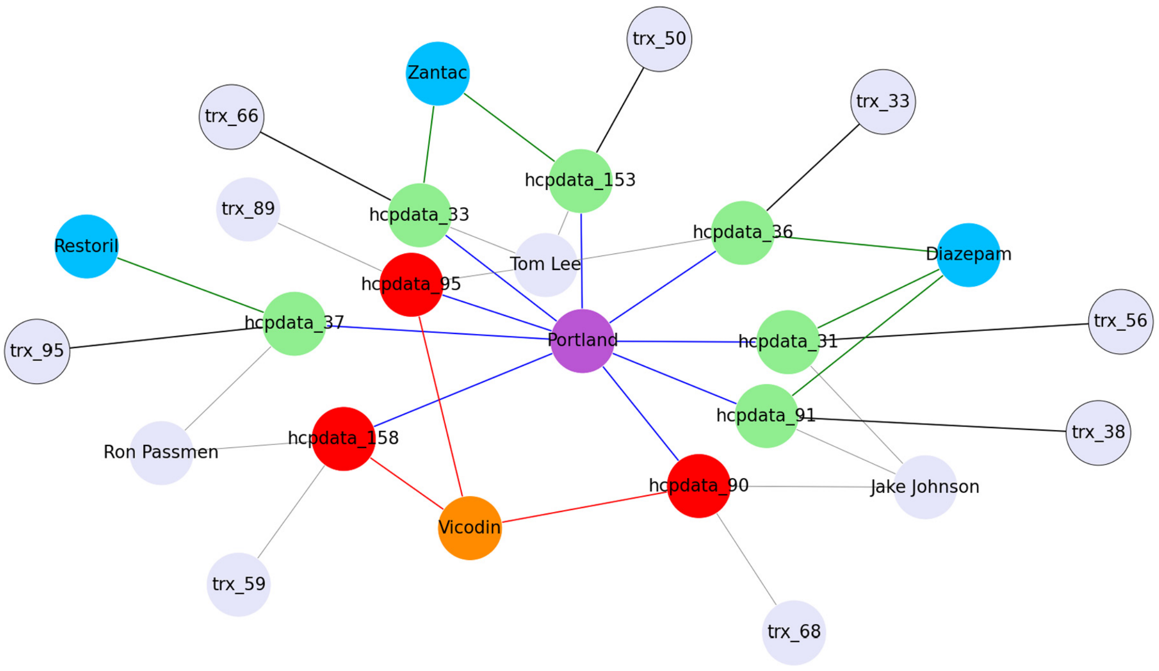
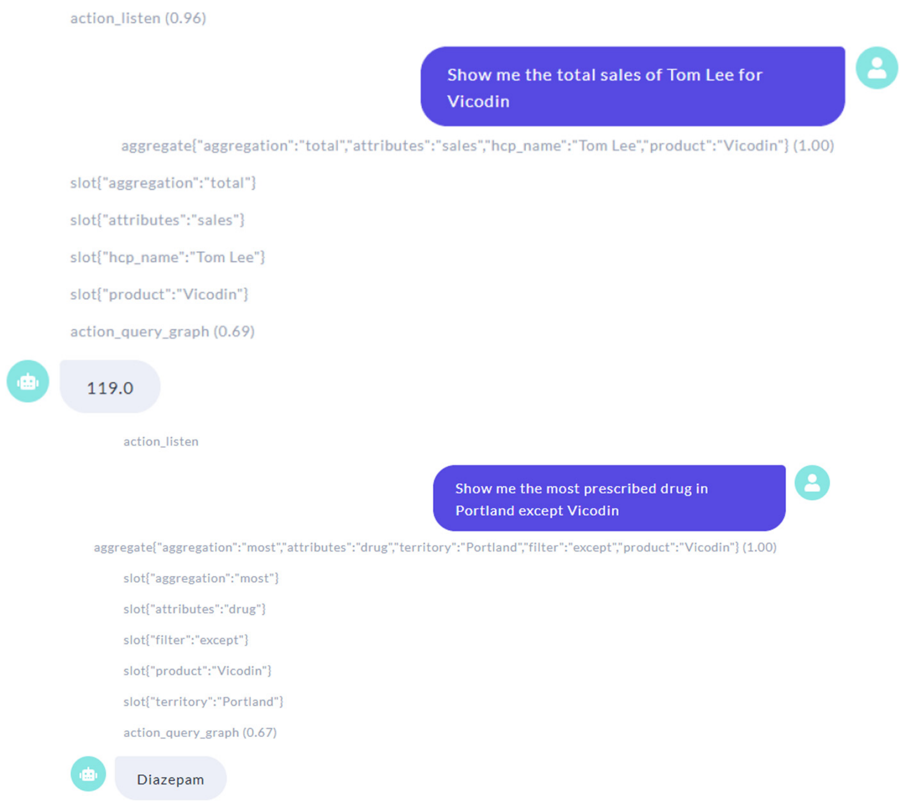


Fig. 5. Subgraph for the query *'Show me the most prescribed drug in Portland except Vicodin?'*



action_listen (0.96)

Show me the total sales of Tom Lee for Vicodin

aggregate{"aggregation":"total","attributes":"sales","hcp_name":"Tom Lee","product":"Vicodin"} (1.00)

slot{"aggregation":"total"}

slot{"attributes":"sales"}

slot{"hcp_name":"Tom Lee"}

slot{"product":"Vicodin"}

action_query_graph (0.69)

119.0

action_listen

Show me the most prescribed drug in Portland except Vicodin

aggregate{"aggregation":"most","attributes":"drug","territory":"Portland","filter":"except","product":"Vicodin"} (1.00)

slot{"aggregation":"most"}

slot{"attributes":"drug"}

slot{"filter":"except"}

slot{"product":"Vicodin"}

slot{"territory":"Portland"}

action_query_graph (0.67)

Diazepam

Fig. 6. Chatbot interface for the questions.

marked in *Green* are the final *primary keys* from which we find the total trx values for each individual product (other than *Vicodin*). The required *trx (sales)* nodes have *Black* boundaries. As can be seen from the figure, the sales value for *Diazepam* adds up to the maximum value of *'127'* thus giving *'Diazepam'* as the result. If we see the total *trx* value for *Vicodin*, it is greater than all the result with the value of *'216'*, however *Vicodin* has been filtered from the answer as was required in the question.

Fig. 6. shows the user interface of RASA for the above questions. We can see that RASA fills all the slots as per extracted information. There is **aggregation** and **filter** slots for queries, **attributes** slots for *Answer_Class*, the *entities* are key-value pairs in slots. The **action_query_graph** indicates the next action to take, which is querying graph. The scores beside it represent the **confidence score** of the above extracted entities and intents. If the confidence score falls below a certain threshold, **fallback condition** is initiated.

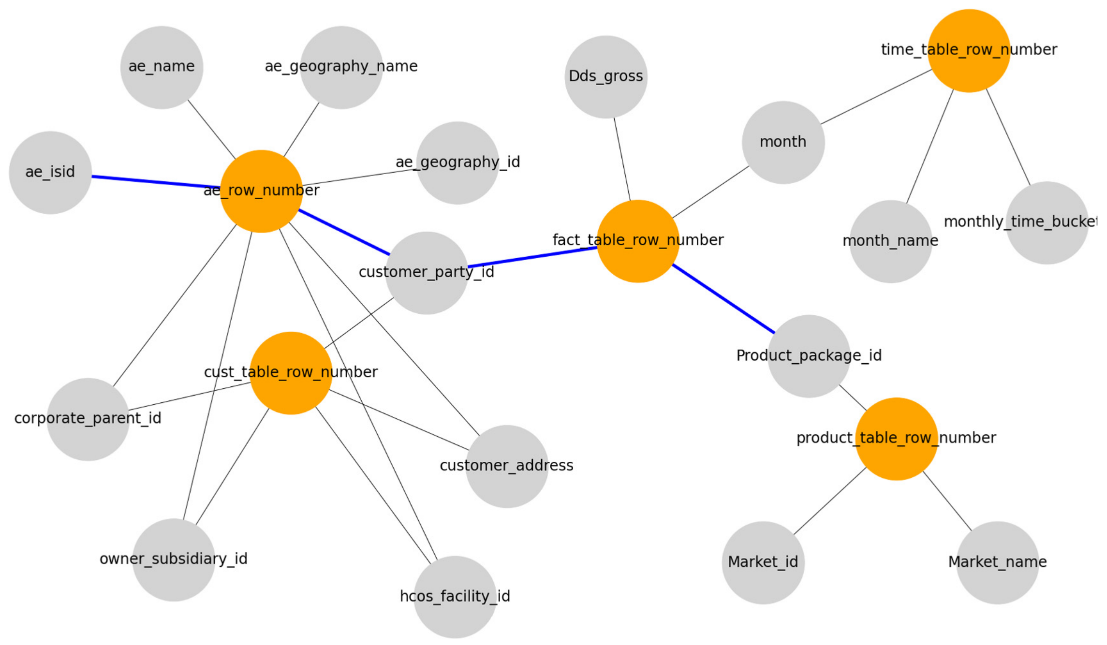


Fig. 7. Shortest path on ontology graph.

5.2. Shortest path approach on ontology graph

As discussed earlier, the shortest path approach in this case is applied on the ontology instead of generating subgraphs and then finding paths.

Let's take the example of a query-

Find all the Product Package IDs associated with the ae_isid ATKINSON?

Calculating the shortest path on the ontology graph gives us the shortest path as (Fig. 7.)-

ae_isid -> ae_row_number -> customer_party_id -> fact_table_row_number -> Product_package_id.

Based on this shortest path obtained, the following SPARQL query gets generated-

```
SELECT ?destination WHERE {
    ?primary2 has_customer_party_id intermediate1.
    ?primary2 has_Product_package_id ?destination.
        {SELECT ?intermediate1 WHERE {
            ?primary1 has_customer_party_id ?intermediate1.
            ?primary1 has_ae_isid AKINSO.}}}
```

5.3. Experimental results

The experiment has been done on the dataset which consists of 11M triples and a smaller subset of the dataset containing 1M triples. The entire architecture has been tested for a total of 255 questions of each of the following types: complex query with no filter and aggregation; complex query with filter; complex query for all the eight aggregation types mentioned above; complex query with filter and aggregation (all eight types); spelling errors; and missing entities, as has been discussed in Section 3 with examples. As is evident, the shortest path approach is just an algorithm which is bound to fetch correct answers if the inputs to it are correct, the accuracy of each type of questions depends on the accuracy with which RASA is able to extract entities and intents from the queries.

While the model is able to answer all queries with no filter or aggregation, spelling errors, filter queries; however, for queries of aggregation types where *Aggregation_Entity* needs to be correctly identified and separated from *Query_Entities*, the accurately answered questions are relatively lesser. The NLU model is failing to extract and separate the *Aggregation_Entity* from *Query_Entities* for question structures it hasn't been trained on. For query forms on which RASA is completely untrained, it sometimes misses to extract entities and intents leading to fallback condition (which occurs in case of missing entities). These can be considered as cases of false negatives. In cases of spelling errors, if the threshold value obtained during entities matching with graph is lesser than the given value, the query goes into fallback. The overall accuracy obtained over *255* questions is *91.76%* with the precision and recall values being *1* and *0.9176*. Table 1 shows individual accuracies, precision, recall and F1 scores obtained for each type of questions.

5.4. Analysis and discussion

To summarize we have presented two methods of graph traversal where

- we traverse a subgraph of the entire graph, containing all the data, in NetworkX or 'traversal on sub-graph' in short
- we traverse the ontology graph which contains the relations between the different classes (and not actual data) and generate dynamic queries for the Stardog graph DB and this can be said as 'traversal on ontology'

Both the methods overcome the burden of traversing the entire graph, thus reducing the time. But the second method has performed better than first one. We have compared the time taken by both 'traversal on sub-graph' and 'traversal on ontology' methods, and see significant improvements with the use of the later approach. This is predictable as the sub-graph contains some data whereas the ontology just contains the skeleton on which the entire graph is built. Table 2 shows the improved performance of the system with the use of Graph Database (DB). For the smaller dataset, as we compare the second method is almost thousand times faster than the first one and for most of the questions, the time taken is just a few seconds even on the huge number of triples of the entire dataset.

The use of RASA along with a graph DB like Stardog to fetch answers to natural language queries has high accuracies and takes very less time. From Table 1, we can see that a certain type of queries has

Table 1. Accuracies for each type of complex queries.

Query Type	No. of queries	Accuracy	Precision	Recall	F1 Score
Queries with no filter or aggregation	25	96	1	0.96	0.979
Queries with filter	25	92	1	0.92	0.958
Queries with aggregation	80	91.25	1	0.9125	0.9542
Queries with filter and aggregation	80	88.75	1	0.8875	0.9404
Queries with spelling errors	25	92	1	0.92	0.958
Queries with missing entities	20	100	1	1	1

Table 2. Time taken in NetworkX vs GraphDB.

Question	Time taken using NetworkX (in seconds)	Time taken in Stardog with small dataset (in seconds)	Time taken in Stardog with entire dataset(in seconds)
Who are the customers in DIABETES in NORTHERN NEW ENGLAND HFAE	312	0.3	3.86
Who are the doctors prescribing in DPP-4 market	208	0.18	3.58
What are the products in HEART FAIL market	390	0.32	0.38
What is the least prescribed product package by customer 652	20.4	0.018	0.022
What is the largest customer party in corporate parent id CP1405190	21.6	0.04	2.52
Who are the top performing customers in NEW YORK METRO HFAE	64.2	0.035	2.55
Show me the total sales for DIABETES for YTD	198.6	0.31	44.07

lesser accuracies than others. We have discussed earlier the reasons behind the lesser accuracy of RASA for those query types. We discuss here the ways in which we plan to improve it.

For *Aggregation* types of queries, where there is a need to separate *Aggregation_Entities* from *Query_Entities*, the way to improve this detection is to include higher number of such queries in training data. The accuracy will improve if RASA is trained on variety of such queries.

We have also discussed cases for lesser accuracies on untrained data. However, we don't want to increase the training data to a large amount. So to improve accuracies on queries which RASA has not seen before, we can train the different components of the RASA. As discussed in the model description in Section 4.3, customized units in pipelines can be used. We can improve the accuracy for unseen queries by training the **Classifier** used in the pipeline on domain specific data. This can improve RASA's the entity recognition and intent classification of NLU on our data.

For spelling errors, different values of threshold can be tried above which spellings can be corrected. We are in the process of identifying the ideal threshold value, for which fallback condition (due to spelling errors) is minimum.

6. Conclusion

We presented in this paper a graph-based approach to answer queries on Structured Data. With the use of Conversational Analytics tool, it behaves as a chatbot system which can fetch answer over multiple tables. The Filter and Aggregation techniques along with the Shortest path approach widens the variety of complex questions that can be answered by the system. Implementing various Filtering and Aggregation functions can be done once we can get to the correct primary keys via the Shortest path. Currently only eight aggregation functions are implemented and a simple Filter function. Various other functions will be implemented further. We will also be aiming to improve performance of RASA to achieve improved accuracies.

We plan to extend this approach from structured to unstructured data, so that questions from various documents and tables can be answered by a single model. The graph modeling will be designed to incorporate unstructured documents to build a single platform for all kinds of data available from which a user may want an answer.

Declarations

Author contribution statement

S. Varma, S. Shivam: Conceived and designed the experiments. S. Biswas, P. Saha: Performed the experiments; Wrote the paper. K. Jalan: Performed the experiments; Analyzed and interpreted the data.

Funding statement

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Data availability statement

The data that has been used is confidential.

Declaration of interests statement

The authors declare no conflict of interest.

Additional information

No additional information is available for this paper.

Acknowledgements

This work has been done in and supported by ZS Associates, Big Data PoD.

References

[1] P. Dasigi, M. Gardner, S. Murty, L. Zettlemoyer, E. Hovy, Iterative search for weakly supervised semantic parsing, in: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Vol. 1 (Long and Short Papers), 2019, pp. 2669–2680.

[2] R. Agarwal, C. Liang, D. Schuurmans, M. Norouzi, Learning to generalize from sparse and underspecified rewards, in: International Conference on Machine Learning, PMLR, 2019, pp. 130–140.

[3] K. Clark, M. Grove, E. Sirin, https://docs.stardog.com/.

[4] J. Berant, A. Chou, R. Frostig, P. Liang, Semantic parsing on freebase from question-answer pairs, in: Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, 2013, pp. 1533–1544.

[5] S.W.t. Yih, M.W. Chang, X. He, J. Gao, Semantic parsing via staged query graph generation: question answering with knowledge base, in: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), 2015.
[6] J. Herzig, P.K. Nowak, T. Müller, F. Piccinno, J.M. Eisenschlos, TAPAS: Weakly supervised table parsing via pre-training. ArXiv preprint, arXiv:2004.02349, 2020.
[7] X. Huang, J. Zhang, D. Li, P. Li, Knowledge graph embedding based question answering, in: Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining, 2019, pp. 105–113.
[8] A. Abujabal, M. Yahya, M. Riedewald, G. Weikum, Automated template generation for question answering over knowledge graphs, in: Proceedings of the 26th International Conference on World Wide Web, 2017, pp. 1191–1200.
[9] W. Zheng, H. Cheng, L. Zou, J.X. Yu, K. Zhao, Natural language question/answering: let users talk with the knowledge graph, in: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, 2017, pp. 217–226.
[10] C. Zhu, K. Ren, X. Liu, H. Wang, Y. Tian, Y. Yu, A graph traversal based approach to answer non-aggregation questions over DBpedia, in: Joint International Semantic Technology Conference, Springer, 2015, pp. 219–234.
[11] X. Lu, S. Pramanik, R. Saha Roy, A. Abujabal, Y. Wang, G. Weikum, Answering complex questions by joining multi-document evidence with quasi knowledge graphs, in: Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, 2019, pp. 105–114.
[12] T. Bocklisch, J. Faulkner, N. Pawlowski, A. Nichol, Rasa: open source language understanding and dialogue management, arXiv preprint, arXiv:1712.05181, 2017.
[13] T. Bunk, D. Varshneya, V. Vlasov, A. Nichol, Diet: lightweight language understanding for dialogue systems, arXiv preprint, arXiv:2004.09936, 2020.
[14] A. Hagberg, P. Swart, D. S Chult, Exploring network structure, dynamics, and function using NetworkX, Technical Report, Los Alamos National Lab. (LANL), Los Alamos, NM, United States, 2008.
[15] M. Musen, S. Tu, M. Horridge, H. Josef, R. Gonçalves, https://protege.stanford.edu/.