CHAIN DOCUMENTATION

TANYA HOWDEN

DIANA BENTAL

LAST UPDATED: 23 JANUARY 2022

TABLE OF CONTENTS

OBJECTIVES OF CHAIN

The CHAIn (Combining Heterogeneous Agencies' Information) system dynamically re-writes queries to databases when mismatches led to query failure.

In general, queries to databases only succeed if they are correctly writing according to the schema of that database, which is only possible if the schema of the database is known in advance. During an emergency response (and, indeed, many other situations), this is plausible in some situations, but emergencies are characterised by their unpredictability and such foresight is not always possible. If one wishes to broadcast a query to all relevant agencies, or to a new agency, or to a previous collaborator who has updated their data sources, for example, a query may fail even if there is relevant data in the data source.

CHAIn sits locally within an agency. When a query to the data sources of that agency fails, CHAIn will use various matching techniques to determine on-the-fly whether there is mismatch between the schema of the query and the schema of the database and, if so, whether there is anything in the database that matches, or approximately matches the query. If so, CHAIn rewrites the query according to the schema of the database and sends back an appropriate response (or responses), together with a score indicating how good the match is and information about the assumptions and approximations made in the match.

SETTING UP CHAIN

1. Download the CHAIn project from Github:

   ```
   git clone https://github.com/CHAIN-HW/CHAINJava.git
   ```

   This creates a directory *CHAINJava*

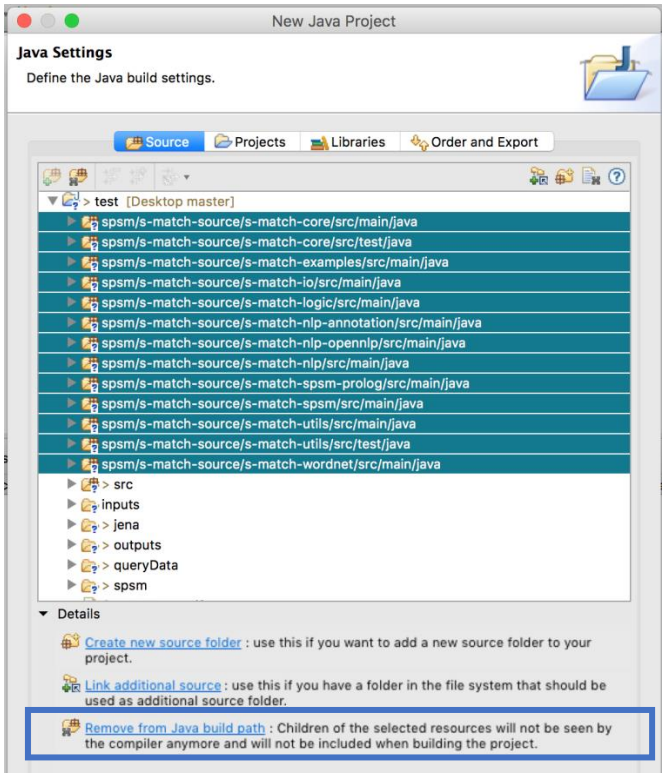
2. Change to directory *CHAINJava/spsm/s-match-source* and run the following four files in this directory *0cloneSMatch, 2buildSMatch, 3createBinary, 4extractBinary.* At the terminal/command prompt type *./* followed by the name of the file,

   ```
   ./0cloneSMatch
   ./2buildSMatch
   ./3createBinary
   ./4extractBinary
   ```

   Doing this should create the following folders in that directory: *s-match-parent, s-match-core, s-match-io, s-match-logic, s-match-nlp, s-match-nlp-annotation, s-match-nlp-opennlp, s-match-spsm, s-match-wordnet, s-match-examples, s-match-utils, s-match-spsm-prolog.*

3. Open Eclipse and select *File > New > Project > Java Project* and enter the name for the project. Uncheck the *Use Default Location* radio button and set the location to point to where this project was downloaded to and click *Next*.

4. In the list of files and packages, select the first 13 packages that all start with the directory *spsm/s-match-source/…* and select the option below to *Remove from Java build path*. This is so that the source files for SPSM are not put into their own package in Eclipse but stay as part of our *src* package.



5. Select the *Libraries* tab and then click on *Add External JARS*, navigate to where this project was downloaded and into *spsm/s-match/lib* and select all the files in this directory (apart from *slf4j-log4j12-1.7.21.jar*) and select *Open*.
6. All the files in *jena/apache-jena-2.12.1/lib* should be added in the same way.
7. The jar file *JWI/edu.mit.jwi_2.4.0_jdk.jar* should also be added in the same way.

8. Then click on the *Add Library* button and make sure that the *JRE System Library* & *Junit 4* libraries have been added to the project.

   Note: If you want to access the libraries imported into the project after set-up then you should right click on the project in the *Package Explorer* on the left-hand side of Eclipse and choose *Build Path/Configure Build Path…*

9. Finally, click on *Finish*. You should now have access to all the source and testing files for this project.

NOTE: Setting up CHAIn on Windows requires Git Bash to be installed (or perhaps another similar shell, though this has not been tested). All the terminal commands used in the process need to be executed from the Git Bash command line as opposed to the Windows default. It is worth noting that Java 8 is required for all the scripts to execute correctly – some of them will throw errors when running Java 11 as well as newer versions.

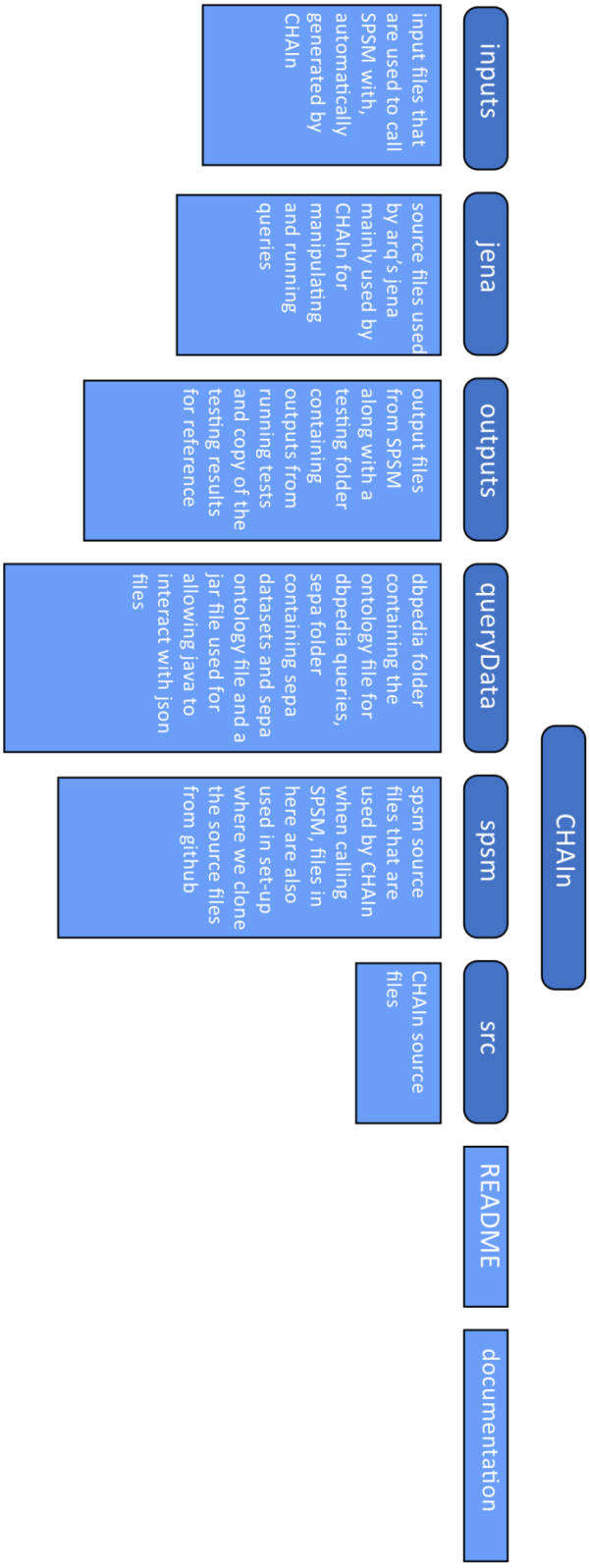
RUNNING FILES IN CHAIN

1. Select the file that you want to run by double clicking on the file in the *Package Explorer* panel on the left.

2. To run, click on the green start button on the top panel of Eclipse and there should be appropriate messages printed to the console to let you know what is happening.

Note: There are some files that will not do anything when you try to run them. You can check what should happen when you run a specific file by looking up that file in this document.

FILE STRUCTURE



TESTING RESULT DOCUMENTS

Files that have names ending Test_Cases.java are tests files and can be found in the tests package. The files write results to a text file in the outputs/testing folder. Most are now structured as JUnit tests with success and failure conditions although some still produce output which needs to be hand checked.

Canonical versions of the test output files are in outputs/testing/copy_of_test_outputs/

BREAKDOWN OF TESTING DOCUMENTS

Running all the tests – *CHAINTests.java*

Task 1 testing documents – *Task1_Test_suite.java*

This task is responsible for testing the core interface with SPSM, i.e. sending a query schema and a set of target schemas to SPSM and returning the list of matches plus its score. Matches are then ranked and filtered by the similarity threshold and any other restrictions.

Task 2 testing documents – *Task_2_Test_suite.java*

This task creates repaired schemas and SPARQL queries from the list of matches created from task 1. It runs the SPARQL queries over the target dataset. The following files make up the functionality for this task:

- *Repair_Schema_Test_Cases.java* (which produces *Schema_Repair_Tests.txt*)
- *Create_Query_Test_Cases.java* (which produces *Create_Queries_Tests.txt*)
- *Run_Query_Test_Cases.java* (which produces *Run_Queries_tests.txt*)

Task 4 testing documents

- *Schema_From_Query_Test_Cases.java* (which produces *Schema_From_Query_Tests.txt*)

Overall CHAIn Functionality

The overall CHAIn functionality involves passing in a SPARQL query which is then processed and attempted to run over the target dataset. If this query does not successfully run (returning no data) then we will extract a schema from this query.

The schema extracted will then be passed into SPSM along with a set of target schemas in order to attain match information which can then be used to create new queries that may be successful. The following file makes up the functionality for this task:

- *Run_CHAIN.java* runs the complete workflow
- *Run_CHAIn_Test_Cases.java* (which produces *Run_CHAIn_Test.txt*)

Experimental tests

Work in progress exploring query repair on examples from the Schema Agnostic Query (SAQ 2015) set.

- *SAQ_Create_Query_Tests.java*

CHAIN RE-IMPLEMENTATION SPECIFICATION

CHAIN RE-IMPLEMENTATION SPECIFICATION

The CHAIN Re-Implementation was divided into four main phases corresponding to four different parts of the CHAIN workflow.

1. SPSM CORE FUNCTIONALITY

Given a query schema and a set of dataset schemas, run SPSM and create a Java data structure which holds the results of the SPSM matches.

PROCESS

- Write the query schema to source.txt
- For each dataset schema
    - Write a dataset schema to target.txt
    - Call spsm
    - Read the result from result.txt
    - If there is a match
        - Add the result to the data structure. The result contains
            - The score for the match
            - The components of the match
- Results are filtered according to a minimum score or number of results required, and sorted by score

- Write the data structure out to two text files
    - in a Prolog–readable format (to maintain backwards compatibility)
    - in a human-readable form (easier to read than Prolog)

INPUT AND OUTPUT DATA STRUCTURES

The schemas are a set of nested terms (see query and dataset examples below). The current format is a series of camel-case phrases, starting with a lowercase letter. The query and dataset schemas should allow nesting even though the current version of CHAIn doesn't need this.

EXAMPLE QUERY SCHEMAS

1. water(timePeriod,geo,measure,resource)
2. auto(brand,name, color)
3. measurement(reporter,node,level,date)

EXAMPLE DATASOURCE SCHEMAS

1. car(year,brand, colour)
2. car(year,brand,colour,price(currency,value))
3. reading(reporter,node,date,water_level)
4. measurement(area,wind_speed,direction)
5. waterBodyPressures(activity,activityCode,affectsGroundwater,assessmentCategory,assessmentParameter,atGepCurClassYear,atGepCurrentPeriod,atGepNextPeriod,atGepPeriodAfterNext,atGepSysdate,comments,dataSource,eiAfterConfidence,eiCurrentConfidence,eiCurrentFailReason,eiNextConfidence,eiNextFailReason,envImprovementAfterNext,envImprovementCurrent,envImprovementNext,hawbDesignationInd,identifiedDate,industrySector,industrySectorCode,isPrimary,locationCode,pressureId,pressureType,protectedAreaId,purpose,purposeCode,swAsset,swmiSector,waterBodyId)

CODE AND TESTS

Match_struc.java
Node.java
Call_spsm.java
Best_match_results.java

SPSM_Test_Cases.java (which produces Call_SPSM_Tests.txt)
Match_Results_Test_Cases.java (which produces Filter_Limit_Tests.txt)
SPSM_Filter_Results_Test_Cases.java (which produces Task_1_Tests.txt)
Task1_Test_Suite.java (which runs the testing files for task 1)

2. CREATE AND EXECUTE A SPARQL QUERY FROM A SCHEMA

Take a schema which may be the result of an SPSM match and use it to create a query and call Jena ARQ

Queries are created using the Jena SPARQL API library
https://jena.apache.org/documentation/query/app_api.html

PARAMETERS TO BE SET

- Whether to use a FROM statement or a TYPE statement for the predicate
- The location of the dataset(s)
- Which ontology mapping file to use
- Maximum number of results to return (or return all results)

DATA STRUCTURES

- Input Query schema represented as Java string
- Input Dataset schema represented as a Java string
- The result of a match from SPSM
- The repaired query (as a tree)
- A data structure which describes the original query and gives mappings from the input query schema to the ontologies used, and mappings from the schema elements to the query data objects
- An ontology mapping file which maps the schema atoms onto full International Resource Identifiers
- Output: result from Jena ARQ
    - Did the query run successfully?
        - If it failed, why?
            - E.g. No dataset found, error in the Sparql …
    - If the query ran, were any results produced?
    - An output file or data structure which contains the results

PROCESS

- Create a repaired schema (or schemas) from the SPSM match result
    - This should be an internal structure, but also printed out as a string for tracing / debugging
- Create a Sparql query
    - This might just be a set of parameters for the Jena API
    - Ideally this should be printed out as a string for tracing / debugging
- Call the Jena API with the Sparql query

- Return the results

CODE AND TESTS

Match_struc.java
Node.java
Repair_Schema.java
Create_Query.java
Run_Query.java
Query_Data.java

Repair_Schema_Test_Cases.java (which produces Schema_Repair_Tests.txt)
Create_Query_Test_Cases.txt (which produces Create_Queries_Tests.txt)
Run_Query_Test_Cases (which produces Run_Queries_Test.txt)
Task2_Test_Suite.java

3. DATA REPAIR

The objective is to take a query which has been generated from a repaired schema but has failed to return any results, and to change the data matching part of the query so as to improve the matching. At present this consists of replacing all object data in the query with variables.

PARAMETERS TO BE SET

- Whether to use a FROM statement or a TYPE statement for the predicate
- The location of the dataset(s)
- Which ontology mapping file to use
- Maximum number of results to return (or return all results)

DATA STRUCTURES

- Input Query schema represented as Java string
- Input Dataset schema represented as a Java string
- The result of a match from SPSM
- The repaired query (as a tree)
- A data structure which describes the original query and gives mappings from the input query schema to the ontologies used, and mappings from the schema elements to the query data objects
- An ontology mapping file which maps the schema atoms onto full International Resource Identifiers

PROCESS

This is the original process for creating a query from the earlier Java versions of the system. The process was extended to include data.

After a SPARQL query has been generated from a schema and run and does not return any data, a version of the query is generated from the same schema with all object fields replaced by variables.

CODE AND TESTS

The repair process is called by *Run_CHAIn.java* and implemented by *Create_Query.java*

Run_CHAIn.java
Match_struc.java
Node.java
Repair_Schema.java
Create_Query.java
Run_Query.java
Query_Data.java
Run_CHAIN.java

Run_CHAIN_Test_Cases – Test 8.0.6 is an explicit test of this process.

Nb there are no tests with data repair in *Run_Query_Test_Cases.java* or *Create_Query_Test_Cases.java*

4. CREATE A CHAIN SCHEMA FROM A SPARQL QUERY

BACKGROUND

The objective is to consider the incoming failed query and extract from it the structure which can be used for matching and repair. This was not part of the original CHAIN system. In principal we could try to extract a schema from any Sparql query by ignoring irrelevant material and picking out the components that are useful for schema matching. But SPARQL is a powerful and expressive language, more expressive than the CHAIn schema language. This leads to two issues

1. Ambiguity – different Sparql queries could lead to the same schema

2. Ambiguity- there is more than one reasonable way to create a schema from a given Sparql query or a dataset, and so different possible schemas could be created depending on how the query is structured. There may not be a consistent (rather than *ad hoc*) way to extract a schema which is effective in all contexts.

For this implementation we have initially specified a subset of Sparql queries from which we expect to extract a consistent and meaningful CHAIn schema. Outside this subset there are other queries from which it is not possible to extract a meaningful schema. To manage the ambiguity issues

1. It is acceptable that different Sparql queries might lead to the same schema if details of the original query can be reconstructed when the repaired query is reconstructed.
2. Parameters can be used to indicate how best to extract the schema from the Sparql query. As exemplars two different styles of query are used for which we expect to create schemas (SEPA style and DBpedia style) and therefore there are two different extraction methods. Other styles may be possible in future.

Initially CHAIn extracts schemas from queries about the properties of a single entity. For example, suppose we want to answer a query about a city where a flood disaster has occurred. We want to know about rivers in the city and the city population.

Sparql Query
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX ont: http://data.ont.org.uk/example/ont/ / PREFIX data: <http://data.ont.org.uk/example/data/ / SELECT * WHERE { ?city rdf:type ont:City ; ont:hasDisaster data:flood ; ont:hasRiver ?river ; ont:hasPopulation ?pop . }
Schema City(hasDisaster, hasRiver, hasPopulation)

The next example queries information about cities of more than 10M people where there has been a flood. The FILTER is ignored and only the central part of the is represented in the schema.

Sparql Query

```
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  ont: http://data.ont.org.uk/example/ont/ /
PREFIX  data: <http://data.ont.org.uk/example/data/ /

SELECT * WHERE
 { ?city rdf:type   ont:City  ;
         ont:hasDisaster   data:flood ;
         ont:hasRiver ?river ;
         ont:hasPopulation   ?pop .
 } FILTER ?pop > 10
```

Schema
City(hasDisaster, hasRiver, hasPopulation)

One example from which we cannot extract a schema is query over multiple linked entities (in database terms this is a JOIN) An example of information about cities which have had floods and also to query the depth of all the rivers. Cities and Rivers are each represented in a separate entity.

Sparql Query

```
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  ont: http://data.ont.org.uk/example/ont/ /
PREFIX  data: <http://data.ont.org.uk/example/data/ /

SELECT * WHERE
 { ?city rdf:type   ont:City  ;
         ont:hasDisaster   data:flood ;
         ont:hasRiver ?river ;
         ont:hasPopulation   ?pop .

   ?river rdf:type   ont:River  ;
         ont:hasDept   ?depth
 }
```

Not clear what schema structure we need! Two separate schemas?

City(hasDisaster, hasRiver, hasPopulation)
River(hasDepth)

PROCESS

For the set of meaningful queries:

- Create schemas for SELECT queries, i.e. queries which return a table of results (not for ASK, CONSTRUCT or DESCRIBE)
- Depending on the query style, the query must contain a FROM or no schema is created
    - For query style 1 (Sepa) the FROM term is used to create the schema predicate
    - For other styles FROM is ignored
- Do not include named graphs
- Only work with simple conjunctive queries about a single entity.So for each RDF triple of <subject, property, object> :
    - The property is never a variable
    - The subject is either a constant (the same constant for the whole query) or else a variable (also the same variable for the query)
    - The object may either be a variable or a constant
    - NB Later extension: A query about more than one subject might need to create more than one schema, or it might use nested schemas.
    - We do not deal with UNION (disjunction)
- Depending on the schema style, the query must contain an rdf:type or no schema is created
    - For query style 2 (DBpedia style) rdf:type forms the CHAIN schema predicate
    - Without rdf:type it is not clear how to form a CHAIN schema predicate
    - For other query styles the type is treated as just another parameter
- Strip off solution modifiers:
    - Projection (i.e. only keeping specific variables), OFFSET/LIMIT, ORDER BY, DISTINCT
    - Strip off value constraints (FILTER)
    - Strip of COUNT
    - Luckily a lot of these modifiers can simply be ignored

DETAILED EXAMPLES – QUERIES OVER THE SEPA DATASET

These queries have a single subject but they don't have a type.

Query	Sparql Query
17	PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> PREFIX sepaw: <http://data.sepa.org.uk/ont/Water#>

	PREFIX sepaidloc: <http://data.sepa.org.uk/id/Location/> PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX sepaidw: <http://data.sepa.org.uk/id/Water/> SELECT * FROM <file:///home/dsb5/CHAIN/chain-may-2017/chain/code/daisy-spsm/queryRespond/datasets/sepa/waterBodyPressures.n3> WHERE { ?id sepaw:dataSource ?dataSource . ?id sepaw:identifiedDate ?identifiedDate . ?id sepaw:affectsGroundwater ?affectsGroundwater . ?id sepaw:waterBodyId ?waterBodyId }
The schema predicate is extracted from dataset name, in the **FROM** part of the query. All the parts of the query should refer to the same **rdf subject** The schema parameters are extracted from the **rdf property** part of the triple (possibly referred to as the predicate) The **rdf object** part of the triple (?dataSource , ?identifiedDate, ?affectsGroundwater, ?waterBodyId are ignored.	
waterBodyPressures(dataSource, identifiedDate, affectsGroundwater, waterBodyId)	

DETAILED EXAMPLES - DBPEDIA QUERIES

These are example Sparql queries that we expect to handle, with a single rdf subject, and a type. The type is used as the schema predicate.

Example Queries	Sparql CHAIn Schema
01	PREFIX dbo: <http://dbpedia.org/ontology/> PREFIX res: <http://dbpedia.org/resource/> PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> SELECT DISTINCT ?uri WHERE { ?uri rdf:type dbo:City . ?uri dbo:isPartOf res:New_Jersey .

	?uri dbo:populationTotal ?inhabitants FILTER (?inhabitants > 100000) }
The FILTER is ignored All the parts of the query should refer to the same **rdf subject** The schema predicate is extracted from the rdf **object** part of the **rdf:type** triple The schema parameters are extracted from the **rdf property** part of the triple (possibly referred to as the predicate) The other **rdf object** parts of the triple (res:New_Jersey, ?inhabitants) are ignored.	
City(isPartOf, populationTotal)	
02	PREFIX yago: <http://dbpedia.org/class/yago/> PREFIX res: <http://dbpedia.org/resource/> PREFIX dbp: <http://dbpedia.org/property/> PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> SELECT DISTINCT ?uri WHERE { ?uri rdf:type yago:WikicatStatesOfTheUnitedStates . ?uri dbp:timezone ?x FILTER (?uri != res:Utah) }
The FILTER is ignored All the parts of the query should refer to the same **rdf subject** The schema predicate is extracted from the rdf **object** part of the **rdf:type** triple The schema parameters are extracted from the **rdf property** part of the triple (possibly referred to as the predicate) The other **rdf object** parts of the triple (?x) are ignored.	
WikicatStatesOfTheUnitedStates(timezone)	
09	PREFIX dbo: <http://dbpedia.org/ontology/> PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> SELECT DISTINCT ?uri WHERE { ?uri rdf:type dbo:ChessPlayer . ?uri dbo:birthPlace ?x . ?uri dbo:deathPlace ?y FILTER (?x = ?y) }
The FILTER is ignored	

All the parts of the query should refer to the same **rdf subject**
The schema predicate is extracted from the rdf **object** part of the **rdf:type** triple
The schema parameters are extracted from the **rdf property** part of the triple (possibly referred to as the predicate)
The other **rdf object** parts of the triple (?x, ?y) are ignored.

CODE AND TESTS

Create_query.Java

Run_query_test_cases.java

Create_Query_Test_Cases.txt (which produces *Create_Queries_Tests.txt*)

DESCRIPTION OF IMPLEMENTATION FILES

MATCH_STRUC.JAVA

Match_struc is the data structure that holds all the information about a source schema. It was originally orginrally intended to hold match information for a source schema but it holds other information such as the repaired query

BASIC FUNCTIONALITY

Data structure that has been created to store the results that are returned after calling SPSM. The structure allows for easy interaction to find out information about the match details with a certain target schema.

DESIGN

This java file represents a class that is responsible for creating an object that is used to store details about a schema/query to allow repairs to take place to successfully query a dataset.

PRIVATE VARIABLES/FIELDS

similarity

This is the variable that stores the similarity value between the source and this current target schema. (double)

dataset

This variable holds the name of the target schema that this structure is representing the results for. (String)

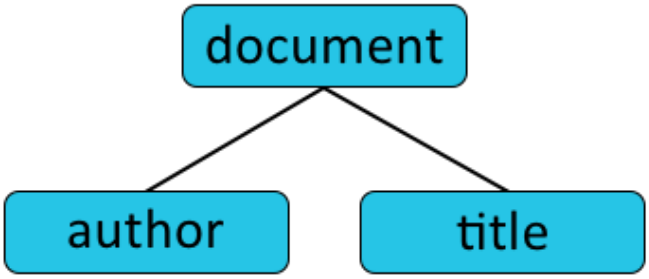
matches

List of arrays containing the match concepts between the source and this current target schema. (ArrayList of String arrays

repairedSchemaTree

Tree structure that represents the repaired schema in a tree form. (Node)

For example: If the schema is *document(title,author)* then the tree will look like the following where *document* is the root node with two child nodes with the values *author* and *title*,



repairedSchema

This variable holds the repaired schema in a format that can easily be read. (String)

numMatches

Stores the total number matches which is calculated by the number of elements in the matches ArrayList. (int)

query

Stores the query that has been produced using the repaired schema (*String*)

METHODS

public Match_Struc (double sim, String targetSchema)

Constructor that is responsible for initialising the Match_Struc variable with the similarity value and the name of the target schema.

public void setSimilarity (double sim)

Sets the similarity of this match to be the value passed in as the parameter, *sim.*

public void setDatasetSchema (String targetSchema)

Sets the target schema to be the String that is passed in as the parameter, *targetSchema*.

public void addMatch (String[] match)

Responsible for adding another match concept to the list of matches stored in the ArrayList<String[]> called *matches.*

public int getNumMatches ()

Returns the total number of matches based on the number of elements inside *matches*.

public double getSimValue ()

Returns the similarity value for that target schema.

public String getDatasetSchema ()

Returns the name of the target schema.

public ArrayList<String[]> getMatches ()

Returns the ArrayList of arrays containing the match concept details.

public String[] getMatchAtIndex (int index)

Returns the specific match concept detail at the specified index passed as a parameter, *index*.

public void setRepairedSchemaTree (Node schemaTree)

Sets the repaired schema tree from the parameter, *schemaTree*.

public void setRepairedSchema (String stringSchema)

Sets the String version of the repaired schema from the parameter, *stringSchema*.

public String getRepairedSchema ()

Returns the String version of the repaired schema.

public Node getRepairedSchemaTree ()

Returns the repaired schema as a tree structure.

public void setQuery(String matchQuery)

Sets the query for this match structure.

public String getQuery()

Returns the query for this structure.

ASSOCIATED TEST FILES

SPSM_Test_Cases.java (which produces *Call_SPSM_Tests.txt*)

*Match_Results_Test_Cases.java (*which produces *Filter_Limit_Test.txt)*

SPSM_Filter_Results_Test_Cases.java (which produces *Task_1_Tests.txt)*

Task1_Test_Suite.java (which runs the testing files for task 1)

USES

Node.java

USED BY

Call_spsm.java

Best_match_results.java

Repair_Schema.java

Create_query.java

Run_query.java

Schema_from_query

RUNNING THIS FILE

-

NODE.JAVA

Simple tree data structure used as part of Match_Struct to store schemas in a tree fornal

BASIC FUNCTIONALITY

Structure that is being used to store the repaired schema tree that consists of a node that has a value and a list of children nodes.

DESIGN

This java file represents a class that is responsible for creating a tree object that is used to store the breakdown of a schema.

PRIVATE VARIABLES/FIELDS

value

Holds the value of the node such as the predicate or parameter name (*String*)

children

The list of nodes that are the children of the current node (*List<Node>*)

METHODS

public Node(String val)

The constructor that creates the node structure and sets the value of that node.

public void addChild(Node child)

Adds the node, *child*, to the list of children nodes.

public String getValue()

Returns the value of the current node as a String.

public List<Node> getChildren()

Returns the list of children nodes.

public ArrayList<String> getChildrenValues()

Returns a list containing the values of all the children nodes.

public Boolean hasChildren()

Returns whether or not the node has any children nodes.

public int getNumChildren()

Returns the total number of children nodes.

public String printTree()

Prints out the tree as a repaired schema.

public String printChildren(List<Node> childrenList)

Prints out the children in the tree to create the repaired schema as a *String*.

public void addToTree(String[] paramParts)

Modifies the tree to add a list of parameters correctly into the current tree.

ASSOCIATED TESTING DOCUMENTS

Repair_Schema_Test_Cases.java (which produces *Schema_Repair_Tests.txt)*

RUNNING THIS FILE

-

USES

.

USED BY

Match_struc

ONTOLOGY_STRUC.JAVA

Holds the ontology information for building SPARQL queries from schemas (and vice versa)

BASIC FUNCTIONALITY

Structure that is used to store the ontology structure being used to create the queries.

DESIGN

This java file represents a class that is responsible for storing the details of an ontology file to allow for other methods to access whilst creating queries.

PRIVATE VARIABLES/FIELDS

name

The name of the ontology structure (*String*) The abbreviated name for the prefix

link

The ontology prefix in full (*String*)

properties

The list of properties associated with the ontology prefix (*String[]*) These are the RDF suffix for each property

METHODS

public Ontology_Struc(String ontName, String ontLink, String[] ontProperties)

Constructor for creating an ontology structure.

public void setName(String ontName)

Sets the name of the structure.

public String getName()

Returns the name of the structure.

public void setLink(String ontLink)

Sets the link for the structure.

public String getLink()

Returns the link of the structure.

public void setProperties(String[] ontProperties)

Sets the properties or key words associated with the structure.

public String[] getProperties()

Returns the list of key words or properties associated with the structure.

ASSOCIATED TESTING DOCUMENTS

Create_Query_Test_Cases.java (which produces *Create_Queries_Tests.txt*)

RUNNING THIS FILE

-

USES

.

USED BY

Create_query.java

Schema_from_query.java?

CALL_SPSM.JAVA

Calls SPSM on source and target schemas

BASIC FUNCTIONALITY

This class is responsible for takes in a target and a set of source schemas and calls SPSM with these schemas. Computes It will then store the results as an ArrayList of Match_Struc objects, with a similarity score for each match.

DESIGN

Inputs

- ArrayList of type Match_Struc objects where you want the results to be stored
- String version of source schema
- String version of target schema

Process

- Calls SPSM with the source and target schema, then picks out the returned results and stores as Match_Struc objects into the ArrayList that was passed in as an input

Outputs

- ArrayList of type Match_Struc objects containing the results from calling SPSM ***OR*** null if there has been a problem with calling SPSM

PRIVATE VARIABLES/FIELDS

targetList

Used temporarily for splitting the target schema String if there are more than one target schema within the String. This allows the application to then call SPSM for each individual target schema. (*String[]*)

METHODS

public ArrayList<Match_Struc> callSPSM (ArrayList<Match_Struc> results, String sourceSchema, String targetSchemas)

Retrieve the schemas from the user, either through the console or as a parameter which gets written to their respective files. Then calls SPSM with the scoruce schema, repeatedly for each target schema.

public ArrayList<Match_Struc> callSPSMOnce (ArrayList<Match_Struc results, String targetSchema)

Makes the call to SPSM using a bash file called *call-spsm.sh.* This should allow for SPSM to run and then write the results out as a serialised object to *serialised-results.ser*. It will then call the appropriate method to read this file and store the results in a Match_Struc object.

public ArrayList<Match_Struc> readSerialisedResults (ArrayList<Match_Struc> results, String targetSchema)

Responsible for reading the serialised object from *serialised-results.ser* and calls the appropriate method to parse the results and stores what is required for CHAIn.

public ArrayList<Match_Struc> parseMatchObject (ArrayList<Match_Struc> results, String targetSchema, IContextMapping<INode> mapping)

Reads the information stored in the object read in (from SPSM) and picks out the data that is required for use by CHAIn.

public String getNodePathString (INode node)

Responsible for taking in a *INode* object (from SPSM) and turns it into something readable so that we can store the match concepts in a convenient way.

ASSOCIATED TESTING DOCUMENTS

SPSM_Test_Cases.java (which produces *Call_SPSM_Tests.txt*)

SPSM_Filter_Results_Test_Cases.java (which produces *Task_1_Tests.txt)*

Task1_Test_Suite.java (which runs the testing files for task 1)

RUNNING THIS FILE

Running this file will call SPSM with the source and target schemas that have been declared in the *main* method and will then print out the results that have been returned from SPSM by printing out the target schema, it's similarity to the source schema and the match concepts.

USES

Match_struc.java

USED BY

Run_CHAIN.java

BEST_MATCH_RESULTS.JAVA

Sorts and filters the schema match results from SPSM. Sorts by similarity score. Filters by similarity score threshold and/or by number of matches required.

BASIC FUNCTIONALITY

Responsible for taking results after calling SPSM and then filtering to ensure each result is over the determined threshold, cut the number of results to the desired *n* and sort them so that the match with the highest similarity is at the top of the list.

DESIGN

Inputs

- ArrayList of type Match_Struc containing a list of matches returned from calling SPSM
- The threshold that you want the similarity value of the results to have as a minimum (where passing in 0 will not filter/limit the results)
- The maximum number of results you want returned (where passing in 0 will not filter/limit the results)

Process

- After taking in the list of results after calling SPSM, we remove any elements in the list that are not equal to or greater than the threshold value and then sort the list so that elements are in descending order based on their similarity value. Finally, the list is restricted in size based on the maximum number of elements that were requested

Outputs

- ArrayList of Match_Struc objects that have been filtered based on the threshold value and limit that has been passed in

PRIVATE VARIABLES/FIELDS

-

METHODS

public ArrayList<Match_Struc> getThresholdAndFilter (ArrayList<Match_Struc> results, Double threshVal, int limNum)

Start off by taking in the results, threshold value and the number of results wanted but if these values haven't been passed in through parameters, then ask the user to enter them through the console. It will then strip off any matches that are lower than the threshold value that has been entered.

public ArrayList<Match_Struc> sortResultingMatches (ArrayList<Match_Struc> filteredRes, int limitNo)

This method will sort the resulting matches after they have been filtered to ensure that only those greater than or equal to the threshold value are left. These matches are sorted so that the matches with the highest similarity value is at the top of the list.

public void displayResults (ArrayList<Match_Struc> res)

This method is used to print out the results after they have been filtered and sorted.

ASSOCIATED TEST FILES

Match_Results_Test_Cases.java (which produces *Filter_Limit_Tests.txt)*

SPSM_Filter_Results_Test_Cases.java (which produces *Task_1_Tests.txt)*

Task1_Test_Suite.java (which runs the testing files for task 1)

RUNNING THIS FILE

Running this file will use the Match_Struc variables in the list of results and filter the list to ensure that anything in the list if over a specified threshold and that there are n number of entries if specified by the user. The *main* method will then print out each element in the list after filtering and sorting the list.

USES

Match_struc.java

REPAIR_SCHEMA.JAVA

Create repaired schemas from the (filtered) results of SPSM

BASIC FUNCTIONALITY

Responsible for taking in an ArrayList of Match_Struc objects and going through each one and creating a repaired schema based on the relation between the original source and target schemas.

DESIGN

Inputs

- ArrayList of Match_Struc objects

Process

- For each of the elements in the list passed in, create a repaired schema based on the details from comparing the source and target schemas after calling SPSM

Outputs

- ArrayList of Match_Struc objects where each element now has a repaired schema attribute/field

PRIVATE VARIABLES/FIELDS

-

METHODS

public ArrayList<Match_Struc> prepare (ArrayList<Match_Struc> matchRes)

Takes in the ArrayList of objects and starts processing them in turn to create a repaired schema.

public Match_Struc repairSchema(Match_Struc matchStructure)

Creates a tree structure of the schema and adds that as a field to the Match_Struc object based on the relation between the source and target schemas. It also adds a String version of the repaired schema as a field to the Match_Struc object.

public Node modifyRepairedTree(Node root, String concept)

Using the relation concept data, the tree is modified to ensure that there are only links to parts of the schema that we think there is a match with. This tree is then returned.

ASSOCIATED TEST FILES

Repair_Schema_Test_Cases.java (which produces *Schema_Repair_Tests.txt*)

Create_Query_Test_Cases.java (which produces *Create_Queries_Tests.txt*)

RUNNING THIS FILE

Running this file will go take in the source and target schemas, call SPSM and then create the repaired schema based on the match details between these two original schemas. The results will be printed to the console.

USES

Match_struc.java

USED BY

Run_Chain.java

CREATE_QUERY.JAVA

Create a SPARQL query from a (repaired) CHAIN schema

BASIC FUNCTIONALITY

Responsible for using the repaired schema to create either a sepa or dbpedia query, this query is then added as a field onto the Match_Struc object structure.

DESIGN

Inputs

- ArrayList of Match_Struc objects with elements containing repaired schemas that you want to create queries for
- The query type as a string (either sepa or dbpedia)
- The directory of the dataset if local as a string otherwise null
- The maximum number of results you want returned

Process

- For each of the elements in the list of Match_Struc objects, we create either a sepa or dbpedia query based on the repaired schema that is stored as a field of a Match_Struc element. This query is then stored as an attribute/field

Outputs

- ArrayList of Match_Struc with elements now containing a query as a string

PRIVATE VARIABLES/FIELDS

-

METHODS

public ArrayList<Match_Struc> createQueries(ArrayList<Match_Struc> matchRes, String queryType, String additionalInfo)

Creates the ontology structure and then creates a query for each of the repaired schemas in the list of Match_Struc objects.

public String createQuery(Type, Match_Struc matchDetails, String datafileDir, int noResults)

Creates one query for one repaired schema, of either sepa or dbpedia type.

public String createSepaQuery(Match_Struc matchDetails, String datafileDir, int noResults)

Creates one local sepa query from the match structure.

public String createDbpediaQuery(Match_Struc matchDetails, int noResults)

Creates one remote dbpedia query from the match structure.

public ArrayList<Ontology_Struc> setupPrefixes(JSONArray jsonArr, ArrayList<Ontology_Struc> ontologies)

Reads ontology file and processes it into Ontology_Struc objects.

public String writePrefix(ArrayList<Ontology_Struc> ontologies)

Writes prefixes for query through reading the Ontology_Struc objects.

public String dataMatching(List<Node> children, ArrayList<Ontology_Struc> ontologies)

Finds what ontology file a specific key word or property is from or relates to.

ASSOCIATED TEST FILES

Create_Query_Test_Cases.txt (which produces *Create_Queries_Tests.txt*)

RUNNING THIS FILE

Running this file will setup a sepa or dbpedia query based on the repaired schema produced from calling SPSM on the target and source schemas. These details are specified in the *main* method. It also prints out the repaired schema and the query string created.

USES

Match_struc.java

Ontology_struc.java

USED BY

Run_CHAIN.java

RUN_QUERY.JAVA

Run a SPARQL query locally or remotely using JENA

BASIC FUNCTIONALITY

Responsible for running either a sepa or dbpedia query.

DESIGN

Inputs

- Match_Struc object containing the query as a string that you want to run
- The type of query as a string (either sepa or dbpedia)
- The directory of the dataset as a string if local otherwise null

Process

- The query stored as a field on the Match_Struc object that has been passed in gets extracted and then the query gets setup and run with the details of this dependent on whether it is a sepa or dbpedia query

Outputs

- Results from running the query or error messages if the query was unable to run are printed to the console

PRIVATE VARIABLES/FIELDS

-

METHODS

public boolean runQuery(Match_Struc current, String queryType, String datasetDir)

Responsible for selecting the correct method for running either a sepa or dbpedia query based on the *queryType* parameter that has been passed in.

public boolean runSepaQuery(String query, String datasetToUseDir, Match_Struc currMatchStruc)

Responsible for running the sepa query that has been passed into this method.

public boolean runDbpediaQuery(String query, Match_Struc currMatchStruc)

Responsible for running the dbpedia query that has been passed into this method.

ASSOCIATED TESTING DOCUMENTS

Run_Query_Test_Cases (which produces *Run_Queries_Test.txt*)

RUNNING THIS FILE

Running this file will run either a sepa or dbpedia query depending on the setup in the *main method.* It also prints out the query that it is trying to run followed by the results if the query has run successfully with data returned.

USES

Match_struc.java

USED BY

Run_CHAIN.java

SCHEMA_FROM_QUERY.JAVA

Extract a CHAIN schema from a SPARQL query

BASIC FUNCTIONALITY

Responsible for taking in either a sepa or dbpedia query that is valid and creating a CHAIn schema for that specific query.

DESIGN

Inputs

- Query as a string
- The type of the query as a string (either sepa or dbpedia)

Process

- Parts of the query are extracted to create a schema represented as a tree which is then stored in a Match_Struc object

Outputs

- Match_Struc containing the details of the schema created from the query that was originally passed in

PRIVATE VARIABLES/FIELDS

-

METHODS

public Match_Struc getSchemaFromQuery(String query, String queryType)

Initial method that gets passed in the query as a string and the type of query it is (either sepa or dbpedia). This information is then used to determine whether we need to parse a sepa or dbpedia query and calls methods accordingly to get this information. Also responsible for returning the result as a Match_Struc object with the schema stored as the repaired schema string and also represented as a tree.

public Match_Struc schemaFromSepaQuery(String query, Match_Struc res)

Responsible for taking in the sepa query and using parts of this query in order to create a schema specific for it. This schema is then added to the *res* parameter as a string and represented as a tree before returning it.

public Match_Struc schemeFromDbpediaQuery(String query, Match_Struc res)

Responsible for taking in the dbpedia and using parts of this query in order to create a schema specific for it. This schema is then added to the *res* parameter as a string and represented as a tree before returning it.

public Node createTreeFromSchemaString(String schemaStr)

Takes in the schema represented as a string and parses this string to create the tree representation of the schema which then gets returned.

ASSOCIATED TESTING DOCUMENTS

Schema_From_Query_Test_Cases.java (which produces *Schema_From_Query_Tests.txt*)

RUNNING THIS FILE

Running this file will create a schema for either a sepa or dbpedia query depending on the setup in the *main method.* It then prints out the schema that has been created.

USES

Match_struc.java

Ontology_stru.javac ?

USED BY

Run_CHAIN.java

RUN_CHAIN.JAVA

Run the complete CHAIN workflow as implemented so far.

BASIC FUNCTIONALITY

Responsible for running the overall CHAIn functionality when passed in a query, target schema, the number of results you want returned from the query, the similarity value threshold and the maximum number of queries you want produced.

Tries the input SPARQL query; if the query fails, extract a schema from the query; run SPSM on the query schema and target schemas; sort and filter the results; create repaired schemas and SPARQL queries and run them.

DESIGN

Inputs

- The query as a string
- The type of query as a string (either sepa or dbpedia)
- The target schema(s) as a string with each schema being separated by a ';'
- The maximum number of results you want printed after running the new queries
- The threshold of the similarity value that you want matches from SPSM to meet the condition of (where passing in 0 will not limit/filter the results)
- The maximum number of matches from SPSM that you want to create queries from (where passing in 0 will not limit/filter the results)
- A PrintWriter that you want the results to be printed out to ***OR*** null

Process

- Start off by trying to run the initial query that has been passed in and if this query does not successfully run with no results returned then it will call SPSM with the schema from this query along with the target schemas to create new queries from this information that will then be run

Outputs

- There are no data structures returned from this, however, there will be messages printed to the console and if you choose to pass in a PrintWriter then a file will be produced with commentary of what happened during runtime. This file can be found in *outputs/testing/Run_CHAIn_Tests.txt*.

PRIVATE VARIABLES/FIELDS

getSchema

The instance of the class Schema_From_Query.java that extracts a schema from a query (*Schema_From_Query*)

spsm

The instance of the class Call_SPSM.java that calls SPSM with two schemas and works out the similarity between the two (*Call_SPSM*)

filterRes

The instance of the class Best_Match_Results.java that filters the results after calling SPSM (*Best_Match_Results*)

repairSchema

The instance of the class Repair_Schema.java that uses the match concept information returned from SPSM in order to create a schema (*Repair_Schema*)

createQuery

The instance of the class Create_Query.java that creates either a sepa or dbpedia query depending on the schema that has been created (*Create_Query*)

runQuery

The instance of the class Run_Query.java that runs either a sepa or dbpedia query (*Run_Query*)

METHODS

public void startCHAIn (String query, String queryType, String targetSchemas, int queryLim, double simThresholdVal, int resLimit, PrintWriter fOut)

This method starts off by trying to run the query that has been passed in and if it runs successfully then returns. If we have an invalid query that cannot be run then we also return. Otherwise we start to repair and once that has been completed we try to run the new queries that have been produced.

public ArrayList<Match_Struc> startRepair(Match_Struc current, String targetSchemas, String queryType, String dataset, int queryLim, double simThresholdVal, int resLimit)

Starts off by calling SPSM with the schema extracted from the initial query along with the target schemas that have been passed in by the user. If SPSM returns no results then we return null otherwise we filter the results, create a repaired schema and then create the associated query for that schema. This information is stored as a list of Match_Struc objects which is then returned as an ArrayList.

public Boolean runRepairedQueries(Match_Struc res, String queryType, String dataset)

This method tries to run the query that is attached to the Match_Struc object that has been passed in where if the query does not successfully run and return results then we return false otherwise return true.

ASSOCIATED TESTING DOCUMENTS

Run_CHAIn_Test_Cases.java (which produces *Run_CHAIn_Tests.txt*)

RUNNING THIS FILE

Running this file will carry out the overall CHAIn functionality that has been described whilst printing messages to the console. If the user chooses to pass in a PrintWriter then extra detail about what CHAIn has done will be printed out to the file that can be found in *outputs/testing/Run_CHAIn_Tests.txt*.

USES

Match_struc

Ontology_Struc

Call_spsm.java

Best_Match_Results.java

Repair_Schema.java

Create_Query.java

Run_Query.java

Schema_From_Query.java

JWI_CALLER.JAVA

BASIC FUNCTIONALITY

Calls the MIT Java WordNet Interface (JWI)

PRIVATE VARIABLES / FIELDS

METHODS

public static IDictionary openDictionary ()

Opens the WordNet dictionary.

public static Set<String> getRelatedWords (IDictionary dict, String wordStr, Set<String> associatedWords)

Given an open WordNet dictionary, and a single word, returns a set of words that are associated with the original word.

An existing (possibly empty) set of associated is also passed in, so that we can build up associations for collections of words as well as individual words.

Gets all lexically related words, synonyms and synset-related words except for ANTONYMs.

ASSOCIATED TESTING DOCUMENTS

JWI_Caller_Test_Cases.java – runs as Junit tests, does not write any test output file

RUNNING THIS FILE

Running this file will take some examples of CHAIN predicates and print all the words associated with each one and the number of associated words found.

USES

Requires the JWI library JAR; and a WordNet database in CHAINJava/Wordnet/wn3.1.dict/dict/

The main program requires String_Parser.

USED BY

Narrow_Down.java

STRING_PARSER.JAVA

BASIC FUNCTIONALITY

Parses the strings used in CHAIN into separate words. Mainly used for CHAIN predicates.

PRIVATE VARIABLES / FIELDS

METHODS

public static ArrayList <String> splitCamelCase(String camelCase)

Split a TitleCase or camelCase string into separate words.

public static ArrayList <String> splitCamelCaseLC(String camelCase)

Split a TitleCase or camelCase string into separate words and return all words in lower case.

public static ArrayList <String> splitSeparators(String separatedWords)

Split a string of words where each word is separated by (one or more) non-alphanumeric characters.

ASSOCIATED TESTING DOCUMENTS

RUNNING THIS FILE

Running this file will take some examples of strings and split each one.

USED BY

Narrow_Down.java

Main program in JWI_Caller

NARROW_DOWN.JAVA

BASIC FUNCTIONALITY

Filter the target schemas so as to return only those which are related to the query schema

PRIVATE VARIABLES / FIELDS

METHODS

public static String narrowDown(String queryHead, String targetSchemas)

Given the head (or predicate) from the query, and one or more target schemas, return only the related target schemas.

ASSOCIATED TESTING DOCUMENTS

RUNNING THIS FILE

Does not run independently.

USES

Requires the JWI library JAR

String_Parser

JWI_Caller

USED BY

Run_CHAIn.java

QUERY_DATA.JAVA

BASIC FUNCTIONALITY

Stores the data from the input query that is needed to reconstruct the data matching parts of the query after schema repair. It stores ontologies and prefixes from the query, and maps from schema headings to data values, i.e. from Sparql properties to Sparql objects. Data values are not part of the schema repair process.

VARIABLES / FIELDS

ArrayList<String> uriObjectValues – a list of data object values which are URIs

ArrayList<Node> literalObjectValues – a list of data object values which are literals (e.g. integers, strings, dates, doubles etc)

HashMap<String, String> prefixToURIMaps – maps a prefix used in query to a full URI

HashMap<String, String> localNameToPrefixMaps – maps a local name in a query to its prefix

HashMap<String, Node> localPropertyNameToLiteralObjectMaps – maps a property name (in the source schema) to its data, if the data is a literal. The Node is a Jena Node which allows literal data to contain modifiers such as the language for a string, or whether the datatype is date / integer / double.

HashMap<String, String> localPropertyNameToURIObjectMaps - maps a property name (in the source schema) to its data, if the data is a URI.

HashMap<String, String> resolvedURItoPrefixAndLocalNameMaps – enables a quick lookup of a fully resolved URI to its component name and prefix

String originalQuery – the input query in a standard format generated by Jena

METHODS

Query_Data(String query) – parses the query and stores the data as listed above

toString() – prints out the data as created, for tracing and debugging

ASSOCIATED TESTING DOCUMENTS

Not independently tested – possibly it should be.

Used in Run_CHAIn_Test_Cases.java; Create_Query_Test_Cases.java; Run_Query_Test_Cases.java

RUNNING THIS FILE

Does not exist independently

USES

Uses JENA libraries to parse the query

USED BY

Create_Query.java

Run_Query.java

Run_CHAIn.java

DESCRIPTION OF TESTING FILES

TESTING RESULT DOCUMENTS

Any file that has a name ending in *Test_Cases.java* is consider a testing file and can be found in the main *src* package in the *tests* folder. These tests will write their results to a text file that can be found in the *outputs/testing* folder. These files have been structured to give details about each individual test including what schemas CHAIn is being run with, what the expected result is and what the actual result is that is returned.

Note: When running the testing files for the first time you might need to select the option to run the file using *JUnit*.

BREAKDOWN OF TESTING DOCUMENTS

Task 1 Testing Documents

This task is responsible for sending a query schema and a set of target schemas to SPSM, which will return the list of matches plus its score and information. This list of matches will then be ranked ensuring that each match in the list meets the similarity value threshold and any other restrictions placed on it. The following files make up the functionality for this task:

- *SPSM_Test_Cases.java* (which produces *Call_SPSM_Tests.txt)*
- *Match_Results_Test_Cases.java* (which produces *Filter_Limit_Tests.txt)*
- *SPSM_Filter_Results_Test_Cases.java* (which produces *Task_1_Tests.txt)*

These files can be run at once by running *Task1_Test_Suite.java*

Task 2 Testing Documents

This task is responsible for creating repaired schemas and SPARQL queries from the list of matches created from task 1. It also provides the implementation for running each of these SPARQL queries over the target dataset. The following files make up the functionality for this task:

- *Repair_Schema_Test_Cases.java* (which produces *Schema_Repair_Tests.txt)*

- *Create_Query_Test_Cases.java* (which produces *Create_Queries_Tests.txt)*
- *Run_Query_Test_Cases.java* (which produces *Run_Queries_Tests.txt*)

These files can be run at once by running *Task2_Test_Suite.java*

Task 4 Testing Documents
This task involves extracting the schema from a SPARQL query, the following file makes up the functionality for this task:

- *Schema_From_Query_Test_Cases.java* (which produces *Schema_From_Query_Tests.txt*)

SPSM_TEST_CASES.JAVA

BASIC FUNCTIONALITY

Responsible for testing the output results from *Call_SPSM.java* which takes in source and target schemas and then calls SPSM before reading in results as a serialised object.

After running this test, the results will have been documented in *outputs/testing/Call_SPSM_Test.txt* where you will be able to see the test number, the target and source schemas that SPSM has been called with, the expected result and the actual result. This test should be hand checked.

Note: These tests will be named with a prefix of 1. For example, test 1.1.1, refers to test 1.1 of SPSM_Test_Cases.java.

EXPECTED RESULTS

TEST 1.1.1

This test calls SPSM with empty source and target schemas, therefore, the size of the results returned is 0.

TEST 1.2.1

This test calls SPSM with a source schema but no target schema, therefore, the size of the results returned is 0.

TEST 1.2.2

This test calls SPSM with a target schema but no source schema, therefore, the size of the results returned is 0.

TEST 1.2.3

This test calls SPSM with the same source and target schema, *author(name)*. Therefore, the similarity value returned should be 1.0 with 2 individual matches.

TEST 1.2.4

This test calls SPSM with a single source schema and three target schemas, where only two of these target schemas have a level of similarity with the source schema. Where *author(name ,document)* has a similarity of 1.0 and 2 individual matches and *reviewAuthor(firstname, lastname ,review)* has a similarity of 0.75 and 2 individual matches.

TEST 1.3.1

This test calls SPSM with a single source schema and two target schemas, however, there are no matches between the source and target schemas. Therefore, the size of the results returned is 0.

TEST 1.3.2

This test calls SPSM with a single source schema and two target schemas where only one of these target schemas matches with the source. So only one of the targets has match data. This target is *author(name)* that has a similarity of 1.0 and 2 individual matches.

TEST 1.3.3

This test calls SPSM with a single source schema and four target schemas where only three of these target schemas matches with the source. Therefore, we have three matches in our list of results. Where *author(name)* has a similarity of 1.0 and 2 individual matches, *paperWriter(firstname, surname, paper)* has a similarity of 0.5 and 2 individual matches and *reviewAuthor(firstname, lastname, review)* has a similarity of 0.75 and 2 individual matches.

TEST 1.4.1

This test calls SPSM with a single source and target schema that has a similarity value of 1.0 and 1 individual match.

TEST 1.4.2

This test calls SPSM with a single source and target schema that do not have any similarity, therefore, the size of the results returned is 0.

TEST 1.4.3

This test calls SPSM with a single source and target schema that do not have any similarity, therefore, the size of the results returned is 0.

TEST 1.4.4

This test calls SPSM with a single source and target schema that have a similarity value of 0.5 and 2 individual matches.

TEST 1.4.5

This test calls SPSM with a single source and target schema, however, this is an example that causes SPSM to crash. Therefore, returning no data to CHAIn so we have null results.

TEST 1.5.1

This test calls SPSM with a single source and target schema where it was expected that there would be a similarity of 1.0 returned, however, a similarity of 0.5 was returned with the expected 5 individual matches.

TEST 1.5.2

This test calls SPSM with a single source and target schema that returns a similarity of 0.6 and 5 individual matches.

TEST 1.5.3

This test calls SPSM with a single source and target schema that do not have a similarity, therefore, the size of the results is 0.

TEST 1.5.4

This test calls SPSM with a single source and target schema, however, this is an example that causes SPSM to crash. Therefore, returning no data to CHAIn so we have null results.

TEST 1.5.5

This test calls SPSM with a single source and target schema, however, this is an example that causes SPSM to crash. Therefore, returning no data to CHAIn so we have null results.

TEST 1.5.6

This test calls SPSM with a single source and target schema where the similarity between the two is 0.625 and 12 individual matches.

TEST 1.5.7

This test calls SPSM with a single source and target schema, however, this is an example that causes SPSM to crash. Therefore, returning no data to CHAIn so we have null results.

TEST 1.5.8

This test calls SPSM with a single source and target schema where there are no similarities between the two, therefore, the size of the results returned is 0.

TEST 1.6.1

This test calls SPSM with a single source and target schema where there is a similarity value of 0.5 and 2 individual matches.

TEST 1.6.2

This test calls SPSM with a single source and target schema where there is a similarity value of 0.5 and 2 individual matches.

TEST 1.6.3

This test calls SPSM with a single source and target schema where there is a similarity value of 0.5 and 2 individual matches.

TEST 1.6.4

This test calls SPSM with a single source and target schema where there is a similarity value of 0.5 and 2 individual matches.

TEST 1.6.5

This test calls SPSM with a single source and target schema where there is a similarity value of 0.5 and 2 individual matches.

TEST 1.6.6

This test calls SPSM with a single source and target schema where there is a similarity value of 0.5 and 2 individual matches.

TEST 1.6.7

This test calls SPSM with a single source and target schema, ~~*however, this is an example that causes SPSM to crash. Therefore, returning no data to CHAIn so we have null results.*~~ This example now runs correctly with 3 individual matches.

MATCH_RESULTS_TEST_CASES.JAVA

BASIC FUNCTIONALITY

Responsible for testing *Best_Match_Results.java* file to make sure that when we pass in the results that we have received from SPSM, we get returned only results that have similarity values over the threshold value that are sorted and that we only have n number of results as requested.

After running this test, the results will have been documented in *outputs/testing/Filter_Limit_Test.txt* where you will be able to see the test number, what the threshold value and limit values are that *Best_Match_Results.java* got called with, the expected result and the actual result. This test should be hand checked.

Note: These tests will be named with a prefix of 2. For example, test 2.1, refers to test 1 of Match_Results_Test_Cases.java.

EXPECTED RESULTS

TEST 2.1 – FAIL WITH LIMIT

This test calls *Best_Match_Results.java* with a threshold value of 1.0 and a limit of 5, therefore, the size of the results is 0 because there are no entries matching this criteria.

TEST 2.2 – MULTIPLE SUCCESSES MATCH

This test calls *Best_Match_Results.java* with a threshold value of 0.6 and a limit of 0 meaning that there is no restriction on the number of results returned. This causes the size of the results to be 2 afterwards meaning that there are 2 entries with a threshold greater than or equal to 0.6.

TEST 2.3 – MULTIPLE SUCCESSES MATCH

This test calls *Best_Match_Results.java* with a threshold value of 0.2 and a limit of 0 meaning that there is no restriction on the number of results returned. This causes the size of the results to be 4 afterwards meaning that there are 4 entries with a threshold greater than or equal to 0.2.

TEST 2.4 – SUCCESS WITH LARGE LIMIT

This test calls *Best_Match_Results.java* with a threshold value of 0.2 and a limit of 5 which causes the size of the results to be 4 afterwards meaning that there are 4 entries with a threshold values greater than or equal to 0.2.

TEST 2.5 – SINGLE FAIL MATCH

This test calls *Best_Match_Results.java* with a threshold value of 0.5 and a limit of 0 meaning that there is no restriction on the number of results returned. This causes the size of the results to be 0 afterwards meaning that there are 0 entries with a threshold greater than or equal to 0.5.

TEST 2.6 – EMPTY MATCHES

This test calls *Best_Match_Results.java* with a threshold value of 0.2 and a limit of 0 meaning that there is no restriction on the number of results returned. This causes the size of the results to be 0 afterwards meaning that there are 0 entries with a threshold greater than or equal to 0.5.

TEST 2.7 – SUCCESS WITH LIMIT

This test calls *Best_Match_Results.java* with a threshold value of 0.2 and a limit of 3 which causes the size of the results to be 3 afterwards meaning that there are at least 3 entries with a threshold value greater than or equal to 0.2.

TEST 2.8 – MULTIPLE FAIL MATCHES

This test calls *Best_Match_Results.java* with a threshold value of 1.0 and a limit of 0 meaning that there is no restriction on the number of results returned. This causes the size of the results to be 0 afterwards meaning that there are 0 entries with a threshold greater than or equal to 1.0.

TEST 2.9 – SINGLE SUCCESS MATCH

This test calls *Best_Match_Results.java* with a threshold value of 0.1 and a limit of 0 meaning that there is no restriction on the number of results returned. This causes the size of the results to be 1 afterwards meaning that there was 1 entry with a threshold value greater than or equal to 0.1.

SPSM_FILTER_RESULTS_TEST_CASES.JAVA

BASIC FUNCTIONALITY

Responsible for testing *Call_SPSM.java* and *Best_Match_Results.java* to make sure that after calling SPSM and filtering the results based on the restrictions entered, we receive the appropriate results afterwards.

After running this test, the results will have been documented in *outputs/testing/Task_1_Tests.txt* where you will be able to see the test number, the source and target schemas that SPSM was called with, what the threshold value and limit values are that *Best_Match_Results.java* got called with, the expected result and the actual result. This test should be hand checked.

Note: These tests will be named with a prefix of 3. For example, test 3.1, refers to test 1 of SPSM_Filter_Results_Test_Cases.java.

EXPECTED RESULTS

TEST 3.1 – SUCCESS SINGLE CALL

This test calls SPSM with a single source and target schema that are the same and filters the results with a threshold value of 0.6 and a limit of 0 (meaning that there are no restrictions on the number of results returned). This causes the size of the results to be 1 meaning that there is only 1 result returned from SPSM that matches the filtering criteria.

TEST 3.2 – FAIL WITH LIMIT

This test calls SPSM with a single source and target schema and filters the results with a threshold of 0.0 and a limit of 2. This causes the size of the results to be 0 meaning that there are no results returned from SPSM that matches the filtering criteria.

TEST 3.3 – SUCCESS MULTI CALL

This test calls SPSM with a single source and 4 target schemas and then filters the results with a threshold value of 0.6 and a limit of 0 (meaning that there are no restrictions on the number of results returned). This causes the size of the results to be 2 meaning that there are 2 results that have been returned from SPSM that matches the filtering criteria.

TEST 3.4 – FAIL SINGLE CALL

This test calls SPSM with a single source and target schema and then filters the results with a threshold value of 0.0 and a limit of 0 (meaning that there are no restrictions on the number of results returned). This causes the size of the results to be 0 meaning that there are 0 results that have been returned from SPSM that matches the filtering criteria.

TEST 3.5 – SUCCESS WITH LIMIT

This test calls SPSM with a single source and 4 target schemas and then filters the results with a threshold value of 0.0 and a limit of 2. This causes the size of the results to be 2 meaning that there are 2 results that have been returned from SPSM that matches the filtering criteria.

REPAIR_SCHEMA_TEST_CASES.JAVA

BASIC FUNCTIONALITY

Responsible for testing the element of CHAIn that after calling SPSM and filtering the results will try to create a repaired schema based on the match data between the source and target schemas.

After running this test, the results will have been documented in *outputs/testing/Schema_Repair_Tests.txt* where you will be able to see the test number, the source and target schemas that SPSM was called with, the expected result and the actual result. This test should be hand checked.

Note: These tests will be named with a prefix of 4. For example, test 4.1.1, refers to test 1.1 of Repair_Schema_Test_Cases.java.

EXPECTED RESULTS

TEST 4.1.1

This test calls SPSM with a single source and target schema with the expected repaired schema to be *car(colour, brand)*. However, this is an example that causes SPSM to crash causing no data to be returned to CHAIn so we have null results.

TEST 4.5.6

This test calls SPSM with a single source and target schema with the expected repaired schema to be *conference(paper(title, document(date(day, month, year), writer(name(first, second))))).* The actual repaired schema that gets generated does match this, however, it should be noted that the parameters are in a slightly different order but still have the same meaning.

TEST 4.5.7

This test calls SPSM with a signle source and target schema with the expected repaired schema to be *conference(paper(title, document(writer(name(first, second))))).* However, this is an example that causes SPSM to crash causing no data to be returned to CHAIn so we have null results.

CREATE_QUERY_TEST_CASES.JAVA

BASIC FUNCTIONALITY

Responsible for testing *Create_Query.java* to ensure that after calling SPSM and repairing the target schema we can create the appropriate sepa or dbpedia query.

After running this test, the results will have been documented in *outputs/testing/Create_Queries_Tests.txt* where you will be able to see the test number, the schema that the query is being created based on, the expected result and the actual result.

Note: These tests will be named with a prefix of 5. For example, test 5.1.1, refers to test 1.1 of Create_Query_Test_Cases.java.

EXPECTED RESULTS

TEST 5.1.1 – SEPA QUERY

This test creates a sepa query based on the schema,

waterBodyPressures(dataSource, identifiedDate, affectsGroundwater, waterBodyId)

This creates the appropriate query based on this schema although it should be noted that the parameters are in a slightly different order in the query that is created in comparison to the expected query.

TEST 5.1.2 – SEPA QUERY

This test creates a sepa query based on the schema,

water(timePeriod, geo, measure, resource)

This creates the appropriate query based on this schema although it should be noted that the parameters are in a slightly different order in the query that is created in comparison to the expected query.

TEST 5.1.3 – SEPA QUERY

This test creates a sepa query based on the schema,

waterBodyMeasures(timePeriod, geo, measure, resource)

This creates the appropriate query based on this schema although it should be noted that the parameters are in a slightly different order in the query that is created in comparison to the expected query.

TEST 5.1.4 – SEPA QUERY

This test creates a sepa query based on the schema,

waterBodyPressures(identifiedDate, waterBodyId, assessmentCategory, source)

This creates the appropriate query based on this schema although it should be noted that the parameters are in a slightly different order in the query that is created in comparison to the expected query.

TEST 5.1.5 – SEPA QUERY

This test creates a sepa query based on the schema,

waterBodyMeasures(waterBodyId, secondaryMeasure, dataSource)

This creates the appropriate query based on this schema although it should be noted that the parameters are in a slightly different order in the query that is created in comparison to the expected query.

TEST 5.1.6 – SEPA QUERY

This test creates a sepa query based on the schema,

surfaceWaterBodies(riverName, associatedGroundwaterId)

This creates the appropriate query based on this schema although it should be noted that the parameters are in a slightly different order in the query that is created in comparison to the expected query.

TEST 5.1.7 – SEPA QUERY

This test creates a sepa query based on the schema,

bathingWaters(catchment, localAuthority, lat, long)

This creates the appropriate query based on this schema although it should be noted that the parameters are in a slightly different order in the query that is created in comparison to the expected query.

TEST 5.1.8 – SEPA QUERY

This test creates a sepa query based on the schema,

surfaceWaterBodies(subBasinDistrict, riverName, altitudeTypology, associatedGroundwaterId)

This creates the appropriate query based on this schema although it should be noted that the parameters are in a slightly different order in the query that is created in comparison to the expected query.

TEST 5.1.9 – SEPA QUERY

This test creates a sepa query based on the schema,

bathingWaters(bathingWaterId)

This creates the appropriate query based on this schema although it should be noted that the parameters are in a slightly different order in the query that is created in comparison to the expected query.

TEST 5.2.1 – DBPEDIA QUERY

This test creates a dbpedia query based on the schema,

City(country, populationTotal)

This creates the appropriate query based on this schema although it should be noted that the parameters are in a slightly different order in the query that is created in comparison to the expected query.

TEST 5.2.2 – DBPEDIA QUERY

This test creates a dbpedia query based on the schema,

Country

This creates the appropriate query based on this schema.

TEST 5.2.3 – DBPEDIA QUERY

This test creates a dbpedia query based on the schema,

Astronaut(nationality)

This creates the appropriate query based on this schema.

TEST 5.2.4 – DBPEDIA QUERY

This test creates a dbpedia query based on the schema,

Mountain(elevation)

This creates the appropriate query based on this schema.

TEST 5.2.5 – DBPEDIA QUERY

This test creates a dbpedia query based on the schema,

Person(occupation, birthPlace)

This creates the appropriate query based on this schema.

TEST 5.2.6 – DBPEDIA QUERY

This test creates a dbpedia query based on the schema,

Person(occupation, instrument)

This creates the appropriate query based on this schema although it should be noted that the parameters are in a slightly different order in the query that is created in comparison to the expected query.

TEST 5.2.7 – DBPEDIA QUERY

This test creates a dbpedia query based on the schema,

Cave(location)

This creates the appropriate query based on this schema.

TEST 5.2.8 – DBPEDIA QUERY

This test creates a dbpedia query based on the schema,

FormulaOneRacer(races)

This creates the appropriate query based on this schema.

TEST 5.2.9 – DBPEDIA QUERY

This test creates a dbpedia query based on the schema,

River(length)

This creates the appropriate query based on this schema.

TEST 5.2.10 – DBPEDIA QUERY

This test creates a dbpedia query based on the schema,

Royalty(parent)

This creates the appropriate query based on this schema although it should be noted that the parameters are in a slightly different order in the query that is created in comparison to the expected query.

TEST 5.2.11 – DBPEDIA QUERY

This test creates a dbpedia query based on the schema,

river(length)

This creates the appropriate query based on this schema which is an empty string because we cannot make a query from this schema.

TEST 5.2.12 – DBPEDIA QUERY

This test creates a dbpedia query based on the schema,

Stream(length)

This creates the appropriate query based on this schema which is an empty string because we cannot make a query from this schema.

TEST 5.2.13 – DBPEDIA QUERY

This test creates a dbpedia query based on the schema,

River(Mountain(elevation))

This creates the appropriate query based on this schema.

RUN_QUERY_TEST_CASES.JAVA

BASIC FUNCTIONALITY

Responsible for testing *Run_Query.java* to ensure that after creating a query, we can run it without any errors.

After running this test, the results will have been documented in *outputs/testing/Run_Queries_Tests.txt* where you will be able to see the test number, the query that is being run and whether the query has run successfully. If the query has not run successfully then there will be an error message printed.

Note: These tests will be named with a prefix of 6. For example, test 6.1.1, refers to test 1.1 of Run_Query_Test_Cases.java.

EXPECTED RESULTS

TEST 6.1.1 – SEPA QUERY

This test successfully runs a sepa query that queries the *waterBodyPressures* data file.

TEST 6.1.2 – SEPA QUERY

This test does not successfully run the query. This is because the application fails to find the dataset being defined in the query, *water.n3*, which is not in the sepa data files so this is expected.

TEST 6.1.3 – SEPA QUERY

This test successfully runs a sepa query that queries the *waterBodyMeasures* data file.

TEST 6.1.4 – SEPA QUERY

This test successfully runs a sepa query that queries the *waterBodyPressures* data file.

TEST 6.1.5 – SEPA QUERY

This test successfully runs a sepa query that queries the *waterBodyMeasures* data file.

TEST 6.1.6 – SEPA QUERY

This test successfully runs a sepa query that queries the *surfaceWaterBodies* data file.

TEST 6.1.7 – SEPA QUERY

This test successfully runs a sepa query that queries the *bathingWaters* data file.

TEST 6.1.8 – SEPA QUERY

This test successfully runs a sepa query that queries the *surfaceWaterBodies* data file.

TEST 6.1.9 – SEPA QUERY

This test successfully runs a sepa query that queries the *bathingWaters* data file.

TEST 6.1.10 – SEPA QUERY

This test does not successfully run the query. This is because the application fails to find the dataset being defined in the query, *waterBodyTemperatures.n3*, which is not in the sepa data files so this is expected.

TEST 6.2.1 – DBPEDIA QUERY

This test successfully runs a remote dbpedia query.

TEST 6.2.2 – DBPEDIA QUERY

This test successfully runs a remote dbpedia query.

TEST 6.2.3 – DBPEDIA QUERY

This test successfully runs a remote dbpedia query.

TEST 6.2.4 – DBPEDIA QUERY

This test successfully runs a remote dbpedia query.

TEST 6.2.5 – DBPEDIA QUERY

This test successfully runs a remote dbpedia query.

TEST 6.2.6 – DBPEDIA QUERY

This test successfully runs a remote dbpedia query.

TEST 6.2.7 – DBPEDIA QUERY

This test successfully runs a remote dbpedia query.

TEST 6.2.8 – DBPEDIA QUERY

This test successfully runs a remote dbpedia query.

TEST 6.2.9 – DBPEDIA QUERY

This test successfully runs a remote dbpedia query.

TEST 6.2.10 – DBPEDIA QUERY

This test successfully runs a remote dbpedia query.

TEST 6.2.11 – DBPEDIA QUERY

This test does not successfully run the query. This is because the query is an empty string, therefore, an error occurs as soon as the query gets parsed.

TEST 6.2.12 – DBPEDIA QUERY

This test does not successfully run the query. This is because the query is an empty string, therefore, an error occurs as soon as the query gets parsed.

TEST 6.2.13 – DBPEDIA QUERY

This test successfully runs a remote dbpedia query.

TEST 6.2.14 – DBPEDIA QUERY

This test does not successfully run the query. This is because the query is an empty string, therefore, an error occurs as soon as the query gets parsed.

TEST 6.2.15 – DBPEDIA QUERY

This test successfully runs a remote dbpedia query.

SCHEMA_FROM_QUERY_TEST_CASES.JAVA

BASIC FUNCTIONALITY

Responsible for testing *Schema_From_Query.java* to ensure that we can create a valid schema from either a sepa or dbpedia query.

After running this test, the results will have been documented in *outputs/testing/Schema_From_Query_Test.txt* where you will be able to see the test number, the query that is getting a schema extracted from, the expected schema and the actual schema extracted from the query.

Note: These tests will be named with a prefix of 7. For example, test 7.1.1, refers to test 1.1 of Schema_From_Query_Test_Cases.java.

EXPECTED RESULTS

TEST 7.1.1

This test creates a schema from a valid sepa query and returns the correct schema.

TEST 7.1.2

This test creates a schema from a valid sepa query and returns the correct schema.

TEST 7.1.3

This test creates a schema from a valid sepa query and returns the correct schema.

TEST 7.1.4

This test creates a schema from a valid sepa query and returns the correct schema.

TEST 7.1.5

This test creates a schema from a valid sepa query and returns the correct schema.

TEST 7.1.6

This test creates a schema from a valid sepa query and returns the correct schema.

TEST 7.1.7

This test creates a schema from a valid sepa query and returns the correct schema.

TEST 7.1.8

This test creates a schema from a valid sepa query and returns the correct schema.

TEST 7.1.9

This test creates a schema from a valid sepa query and returns the correct schema.

TEST 7.1.10

This test attempts to create a schema from an invalid sepa query (one of the prefixes are missing) and therefore returns null as expected.

TEST 7.2.1

This test creates a schema from a valid dbpedia query and returns the correct schema.

TEST 7.2.2

This test creates a schema from a valid dbpedia query and returns the correct schema.

TEST 7.2.3

This test creates a schema from a valid dbpedia query and returns the correct schema.

TEST 7.2.4

This test creates a schema from a valid dbpedia query and returns the correct schema.

TEST 7.2.5

This test creates a schema from a valid dbpedia query and returns the correct schema.

TEST 7.2.6

This test creates a schema from a valid dbpedia query and returns the correct schema.

TEST 7.2.7

This test creates a schema from a valid dbpedia query and returns the correct schema.

TEST 7.2.8

This test creates a schema from a valid dbpedia query and returns the correct schema.

TEST 7.2.9

This test creates a schema from a valid dbpedia query and returns the correct schema.

TEST 7.2.10

This test creates a schema from a valid dbpedia query and returns the correct schema.

TEST 7.2.11

This test creates a schema from a valid dbpedia query and returns the correct schema. Note that the query passed in is an empty string, therefore, the schema should be an empty string.

TEST 7.2.12

This test creates a schema from a valid dbpedia query and returns the correct schema.

TEST 7.2.13

This test attempts to create a schema from an invalid dbpedia query (one of the prefixes are missing) and therefore returns null as expected.

RUN_CHAIN_TEST_CASES.JAVA

BASIC FUNCTIONALITY

Responsible for testing *Run_CHAIn.java* to ensure that we can run CHAIn given several different scenarios such as an invalid query or several target schemas to match against the schema extracted from the original query.

After running this test, the results will have been documented in *outputs/testing/Run_CHAIn_Tests.txt* where you will be able to see the test number, the original query that CHAIn has been called with including whether it successfully runs and the steps that CHAIn takes depending on this including any new queries produced and whether they are successful.

Note: These tests will be named with a prefix of 8. For example, test 8.1, refers to test 1 of Run_CHAIn_Test_Cases.java.

EXPECTED RESULTS

TEST 8.1

This test takes in a basic sepa query that doesn't return any results, which after passing into CHAIn results in a new query being created that runs successfully.

TEST 8.2

This test takes in a sepa query; however, this query is invalid meaning that CHAIn terminates immediately after trying to run it.

TEST 8.3

This test takes in a basic sepa query that returns no results, however, after attempting to repair and create new queries we get zero matches from SPSM meaning that no new queries are created.

TEST 8.4

This test takes in a basic dbpedia query that initially returns no results, after running CHAIn with this query we now have a new query that does run successfully and returns results.

TEST 8.5

This test takes in a basic dbpedia query that already runs successfully, meaning that CHAIn terminates immediately after running this initial query.

TEST 8.6

This test takes in a dbpedia query that returns no results, after running CHAIn with this query we now have two new queries where only one of these queries returns results.