NaCaGraph: Visualizing Event Graph for Natural Catastrophes from News Articles – A Project Report

Chong Shen* and **Li Lin*** and **Xiru Tong***
Institute for Natural Language Processing, University of Stuttgart
{chong.shen,li.lin,xiru.tong}@ims.uni-stuttgart.de

Abstract

Recent years have seen increasing number of natural catastrophes, causing loss of life and damage of property. To facilitate the analysis and management of natural catastrophes, it is necessary to develop a system that can collect, visualize and query relevant information about natural catastrophes from various sources. We propose NaCaGraph, a pipeline-based system that scrapes news articles from web pages, extracts events from the articles and visualizes them in a graph database. The system also enables the retrieval of structured information via efficient querying techniques. This project report provides a concise overview of our system and show the relation between our approach and the common ground.[1]

1 Introduction

Natural catastrophes such as drought, earthquake and floods have caused increasing loss in the past 50 years, posing significant risks on the global economy (Gavazzi et al., 2023). This elevates the need of efficient disaster management systems that collect, store, retrieve and visualize data to help better prepare for natural disasters, mitigate their influence and recover from the damages (Khan et al., 2023). Each of the steps requires the adoption of advanced text technologies.

Due to the explosive emergence of data on the internet, it is unrealistic to copy and paste relevant data page by page. To accelerate the data collection, tools and methods that automatically scrape data from web pages have emerged. A simple approach is to parse an HTML page and retrieve the page content using regular expressions or query languages such as XPath (Clark et al., 1999). Alternatively, data can be obtained by web crawlers and scrapers. A web scraper is a program that scans through websites, downloads semi-structured contents, extract relevant data and save it to a database. There are various out-of-box scrapers, such as *Scrapy* (Kouzis-Loukas, 2016) and *Beautiful Soup* (Zheng et al., 2015). In this project, we use the latter due to our previous experience with it.

News articles are an essential source of information about natural catastrophes. However, they are typically in the form of unstructured data with a large amount of noise which makes them inefficient to retrieve, visualize and analyze. Primarily, each news article is a single document, which omits the relations between articles. Furthermore, the linguistic differences in news texts produced by different authors make the analysis even harder. In contrast, events are a more straightforward way to describe situations during natural catastrophes (Verma et al., 2011). Intuitively, an event indicates an occurrence of an action or state change that contains information about *who did what to whom? where, when, and how?* (Li et al., 2022).

In this project, we aim to work on the following questions: (1) How do different components of an event distribute in different articles?, and (2) What are the potential inter-article and inter-topic relationships regarding the components of an event? To answer these questions, we design a pipeline-based system that scrapes news articles from WikiNews web pages, extracts events about natural catastrophes under 7 topics and visualizes them in a database that enables retrieval of diverse information via efficient querying. Figure 1 provides a graphical overview of this project. The detailed workflow can be found in Appendix A.1.

2 Related Works

Web scrapers are widely used for the data collection. *Beautiful Soup* is a powerful python library for extracting, analyzing, and editing information on web pages (Zheng et al., 2015). It generates a DOM tree for the given HTML file and obtains the

*Equal contribution
[1]Code is available at https://github.com/st143575/NaCaGraph.

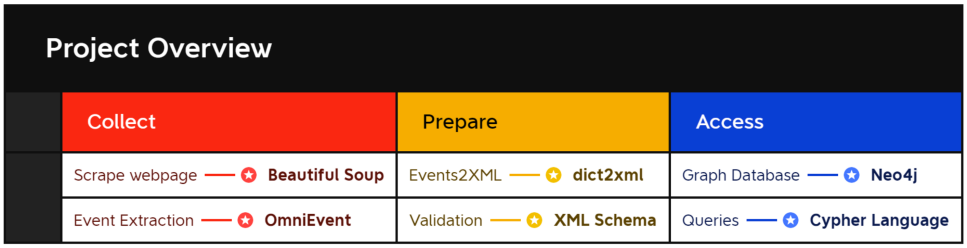

Figure 1: Project Overview.

data by searching the tags and contents in the file.

Li et al. (2022) define an *event* as a compositional structure of four elements: (1) *Event trigger*: the core unit that causes the state change, e.g., the predicate of a sentence; (2) *Event type*: the category to which the event belongs; (3) *Argument*: the main attribute of an event, including entities, non-entity participants, time, place etc.; (4) *Argument role*: the role played by the argument in an event.

Extensible Markup Language (XML) is a markup language for storing and transmitting data (Banzal, 2020). It consists of a declaration, elements, attributes and other characters. An XML file can be validated by an *XML Schema* (Campbell et al., 2003), defining the content of XML files.

Neo4j[2] is a graph database built to store and retrieve connected data. It represents data records as nodes and store their relationships as directed edges explicitly. Nodes and edges can have attributes. Neo4j database can be queried using the *Cypher* language.

3 Data

We demonstrate our system using news articles from the English WikiNews web page.[3] Wikinews is a free-content news platform that creates news articles reported on mainstream media sources and publishes original reports from the public (Ballas, 2006). We select the following 7 topics about common natural catastrophes: *climate change, drought, earthquake, forest fire, greenhouse, high temperature, wildfire.*

4 Methods

This section provides a systematic overview of our system in a pipeline with three phases: **collect**, **prepare** and **access**. The event extraction problem is formulated as a technical extension that connects the first two phases.

[2] https://neo4j.com/

[3] https://en.wikinews.org/wiki/Main_Page

4.1 Collect

In this phase, we aim to obtain desired news articles from web pages and preprocess the collected data. The page regions located by the commands in the following steps are shown in Appendix A.2.

Step 1. We always put the legality in the first place. Before the collection, we check the scraping authorization of the WikiNews web page in *https://en.wikinews.org/robots.txt* to make sure that our scraping actions comply with the rules.

Step 2. We select 7 topics about natural catastrophes, each represented by a category name (Section 3). We search with the 7 category names in WikiNews and copy the URLs of the resulting pages to a TXT file, one URL per line. Next, we send requests to the server using the 7 URLs and retrieve the HTML content of the first 100 pages for each URL. To make the content more readable and accessible, we then parse the HTML content using *Beautiful Soup*. After that, we manually identify the elements in the HTML tree which correspond to the target region in the web page by getting the entire region of the resulting page (Figure 5) and finding the regions of all single results (Figure 6). For each article, we find its metadata (Figure 7) and extract the text span describing the number of words from the metadata. After that, we extract the headline of non-empty articles and create a mapping from the article IDs to the metadata extracted from those articles.

Step 3. Since we aim to model the publish date in our database, we keep only articles with an explicit publish date (Figure 8) and get the content of those articles (Figure 9).

Step 4. To facilitate the event extraction in the next step, we preprocess the scraped articles as follows. First, we add a full stop to the end of the news headline of each article. We then concatenate the headline with the main text, such that sentences

are split by a whitespace. After that, we write the result of each article under each topic to a TXT file.

4.2 Extension: Event Extraction

Based on the event structure defined by (Li et al., 2022) , we extract events from sentences by identifying event triggers and their types, as well as arguments and their roles (see Appendix A.3). We first adopt *OneIE* (Lin et al., 2020), a joint information extraction system trained on ACE05 dataset with 33 event types and 22 argument roles. However, we finally choose the results extracted by a sequence-to-sequence event extractor via *OmniEvent*[4] due to its better domain matching, which attributes to the larger label space in its training set (168 event types and 35 argument roles).

4.3 Prepare

In this phase, we convert the extracted events to the XML format and validate them using an XML schema.

4.3.1 Event2XML

Step 1. We first get the topic by extracting the category name from the file name. Then, we load the Pickle files that contain the extracted events for each topic into a python dictionary. Each item in the dictionary is a mapping from the document ID to a list of structured event records. Due to the limitation on the number of nodes in the database, we limit the number of items by 80.

Step 2. We postprocess the dictionary so that it conforms with the XML structure. Specifically, we merge events into a single dictionary, extract the publish date from the scraping results and merge it into the dictionary. Subsequently, we add a title to each article, extract the topic ID from the scraping results and add the publish date to the dictionary. After substituting topic IDs for article tags, we create a new dictionary that maps the key *category* to the dictionary mentioned above and convert the final dictionary to the XML format using the *dict2xml* package.

Step 3. We modify the article IDs so that they start with the corresponding category IDs. Eventually, the results are written to an XML file for each topic.

[4] https://github.com/THU-KEG/OmniEvent

4.3.2 XML Schema

Each XML document has to be well-formed, namely, adhere to syntactic rules. Optionally, it has to be valid to additional rules defined in a document grammar or schema. We design an XML schema for the validation of all the XML files.

Specifically, we define the namespace for our schema. Furthermore, we define 6 simple elements: *date*, *mention*, *role*, *trigger*, *type* and *title*, as well as 4 attributes: *topic*, *cid*, *aid* and *url*. Then, we define 4 complex elements: *argument*, *event*, *article* and *category*. The *argument* element has an optional sequence of a *mention* element and a *role* element. The element *event* contains a sequence of unbounded *argument* elements, a *trigger* element and a *type* element. The element *article* is comprised of sequence of a *date* element, unbounded *event* elements and a *title* element, as well as an *aid* attribute. The element *category* consists of a sequence of unbounded *article* elements, a *topic* attribute and a *cid* attribute.

We validate our XML files using *XmlLint*. All the XML files pass the validation.

4.4 Access

In this phase, we create a *Neo4j* database from the XML files and design a graph structure for the visualization. We also demonstrate the accessibility of our database via multiple querying examples.

4.4.1 Neo4j Database

Recall that the XML file has a tree structure (see Appendix A.4). Each of our XML files has the root node *category* with an attribute *topic*, followed by multiple *article* nodes. Each *article* node consists of a *date* node, a *title* node and multiple *event* nodes. Each *event* node contains a *trigger* node, an *event_type* node and multiple *argument* nodes. Each *argument* node has an *argument_word* node and an *argument_role* node. We traverse the tree in a breadth-first manner and convert our XML documents to a Neo4j database.

First, we get the first node of the current layer (i.e. *category* in the first layer), get its topic and create the node *topic*. Then, we get the children of the current node (i.e. *article* for the root node) one by one, read their attributes and create those nodes and their relationships in the database (i.e. *article*, *year* in this case). We repeat the steps above layer by layer, until reaching the leaf nodes.

The most crucial part of the graph design is the design of relationships between nodes. In other

*(29,350)

Nodes	Property keys		
Topic	type	text	
Year	type	text	
Article	type	text	
idTrigger	type	text	id
Trigger	type	text	
idArgument	type	text	id
Argument	type	text	
Event_type	type	text	
Role_label	type	text	

Figure 2: Nodes defined in our database.

words, how to define relationships between nodes with different topics and articles via the event components? How to deal with nodes with the same text content? For example, is it better to connect the node *article* to the node *trigger* using a relationship *event_type*, or treat *event_type* as a node and connect it to the nodes *article* and *trigger* through the relationship *relation*, respectively? To answer these questions, we define 9 nodes and 11 relationships as shown in Figures 2 and 3. The resulting graph structure is illustrated in Appendix A.5.

4.4.2 Query

The querying of the information from the Neo4j database is accomplished using the *Cypher* language. Examples are shown in Appendix A.6.

5 Discussion on the question

Question: Describe what the database would have to look like, if you would have used a RDBMS. Would the underlying data model change? What would a query look like?

Answer: Neo4j is a graph database based on the graph theory. Data are represented as nodes and their relationships as directed edges. In contrast, a RDBMS maintains relational databases based on the relational model (Codd, 1970). Data are presented as a collection of tables with rows and columns. Each row is identified by a unique *primary key*. The relationships in the graph database are not directly implemented in the relational database, but need to be inferred by e.g. *foreign keys*. Hence, our data would be represented as tables which take *Event* as the relation name and take the nodes in Figure 2 as attributes. Note that the term *attribute* refers to the name of the role played by a domain in a relation. An example is shown in Appendix A.7.

*(91,369)

Relationships	Property keys	
published_time	type	text
published_article	type	
has_trigger	type	text
has_argument	type	text
has_event_type	type	id
tr_of	type	text
has_role	type	id
event_for	type	
event_for_idtr	type	
role_for	type	
role_for_idarg	type	

Figure 3: Relationships defined in our database.

Similar to Neo4j, data in a relational database can also be queried and modified by query languages. A common query language based on relational algebra is the *Structured Query Language (SQL)*. An SQL query starts with the entry of the table(s) to be returned and applies set operations and relational operations on the table(s). However, querying in a relational database would be much slower than in a graph database due to the lack of explicitly stored relationships (i.e. directed edges).

For instance, the *Cypher* query in Figure 13 is comparable with the following SQL query:

```
SELECT p, a, n
FROM Topic as p, Article AS a, idTrigger
     AS n
WHERE a.event_type = "type:catastrophe"
AND p.publish_time > 2020
```

6 Conclusions

In this project, we aim to help facilitate the analysis and management of natural catastrophes using text technologies. We propose NACAGRAPH, a pipeline-based system that scrapes news articles from the WikiNews web page, extract events under 7 topics about natural catastrophes and create a graphical database that enables the visualization and querying of the events and the relationships between their components. In this report, we provide a systematic overview of the workflow of our system, including web scraping and preprocessing, event extraction, XML creation and validation, database creation and data querying. Our approach can be applied to news articles of other topics, in other languages and to data from other modalities.

Ethics Statement

WikiNews is a free-content news platform. News articles on Wikinews are created by volunteers with various backgrounds and can be modified by any users. The content of the articles may be biased and not factual and should not be interpreted as necessarily representing the stance of the authors of this report.

Acknowledgements

We would like to thank Doctor Kerstin Jung and other reviewers for valuable discussions.

References

Alexios Ballas. 2006. Wikinews: A world flattener? *The Information Systems Student Journal*, 1:7–10.

Shashi Banzal. 2020. *XML Basics*. Mercury Learning and Information.

Charles E Campbell, Andrew Eisenberg, and Jim Melton. 2003. Xml schema. *ACM SIGMOD Record*, 32(2):96–101.

James Clark, Steve DeRose, et al. 1999. Xml path language (xpath).

Edgar F Codd. 1970. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

Paolo Gavazzi, Renata Dobrucka, Stephan Haubold, and Robert Przekop. 2023. The impact of climate change on selected areas of the world economy. *Research on Enterprise in Modern Economy theory and practice*, 1(36):4–16.

Saad Mazhar Khan, Imran Shafi, Wasi Haider Butt, Isabel de la Torre Diez, Miguel Angel López Flores, Juan Castanedo Galán, and Imran Ashraf. 2023. A systematic review of disaster management systems: Approaches, challenges, and future directions. *Land*, 12(8):1514.

Dimitrios Kouzis-Loukas. 2016. *Learning scrapy*. Packt Publishing Livery Place.

Qian Li, Jianxin Li, Jiawei Sheng, Shiyao Cui, Jia Wu, Yiming Hei, Hao Peng, Shu Guo, Lihong Wang, Amin Beheshti, et al. 2022. A survey on deep learning event extraction: Approaches and applications. *IEEE Transactions on Neural Networks and Learning Systems*.

Ying Lin, Heng Ji, Fei Huang, and Lingfei Wu. 2020. A joint neural model for information extraction with global features. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 7999–8009.

Sudha Verma, Sarah Vieweg, William Corvey, Leysia Palen, James Martin, Martha Palmer, Aaron Schram, and Kenneth Anderson. 2011. Natural language processing to the rescue? extracting" situational awareness" tweets during mass emergency. In *Proceedings of the International AAAI Conference on Web and Social Media*, volume 5, pages 385–392.

Chunmei Zheng, Guomei He, and Zuojie Peng. 2015. A study of web information extraction technology based on beautiful soup. *J. Comput.*, 10(6):381–387.

A Appendix

A.1 Workflow

The entire workflow of our system, together with the input and outputs of each step, are illustrated in Figure 4.

A.2 Location of web page regions

The regions located by the commands in Section 4.1 is shown in Figures 5 to 9.

A.3 Event Extraction

Figure 10 shows an example of an event extracted from an input sentence.

A.4 XML Tree

Figure 11 illustrates the tree structure of our XML documents.

A.5 Graph structure in the database

Figure 12 shows the desgined graph structure in our database.

A.6 Query Examples

Figures 13 to 17 show examples of queries for our database. In each figure, the query is on the top and the result is on the bottom.

A.7 Example: Relational Database

Table 1 illustrates an example of a relational database.

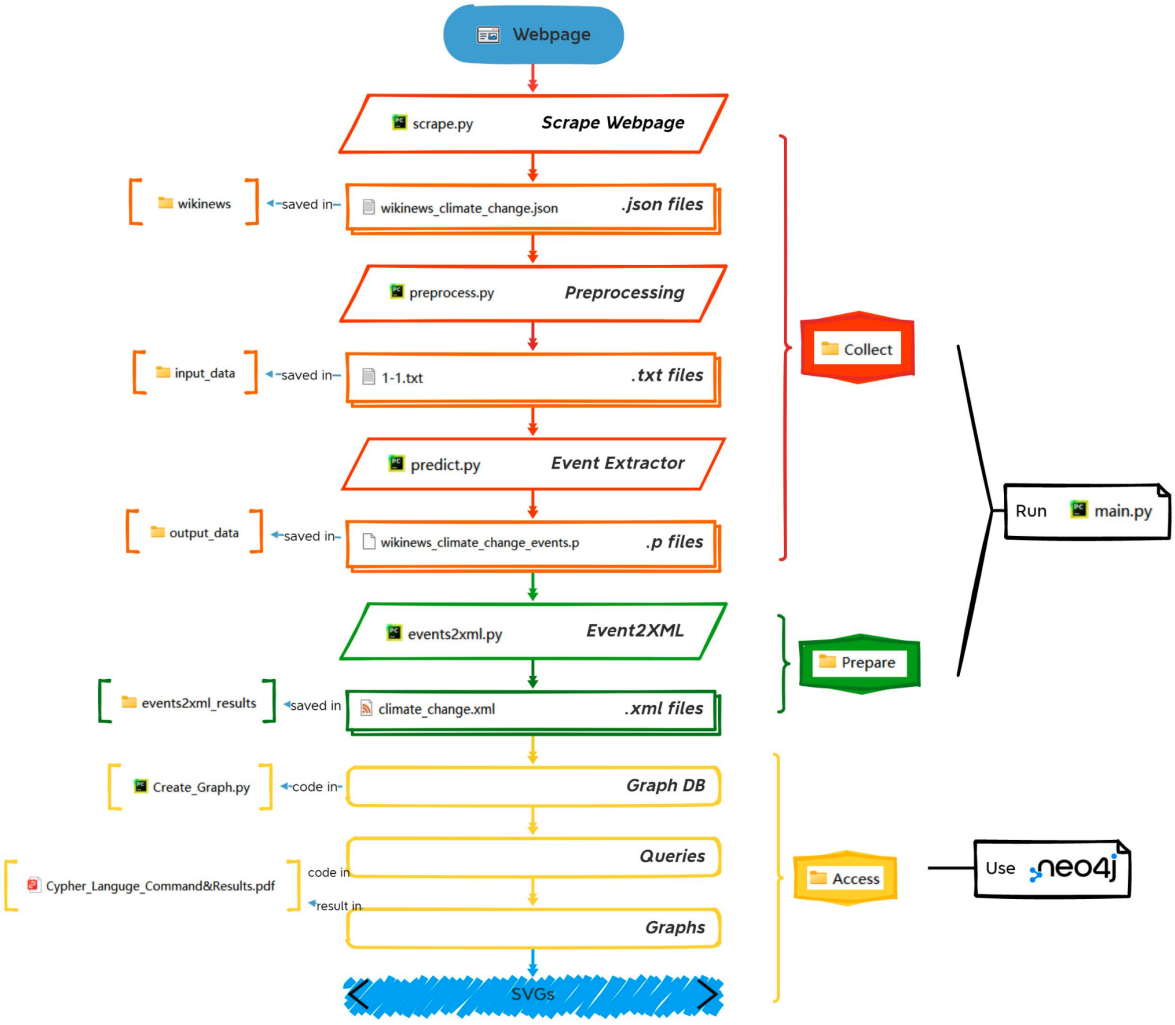

Figure 4: Workflow of NACAGRAPH.

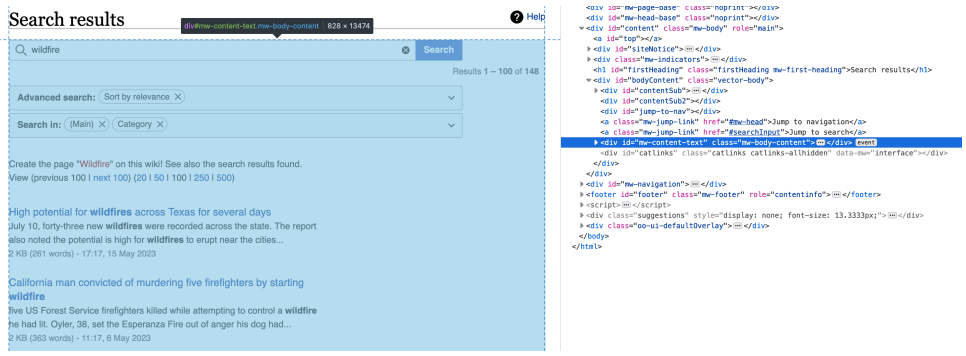

Figure 5: results = soup.find(id="mw-content-text")

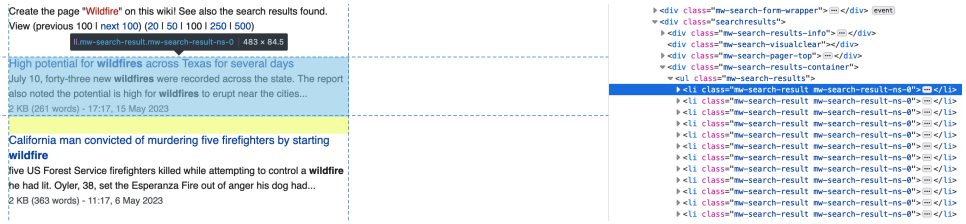

Figure 6: results = soup.find(id="mw-search-result mw-search-result-ns-0")

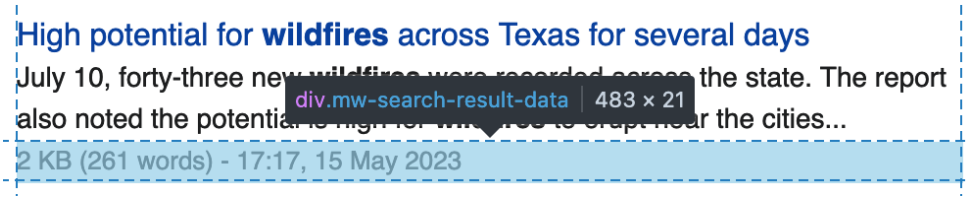

Figure 7: metadata = article.find("div", class ="mw-search-result-data")

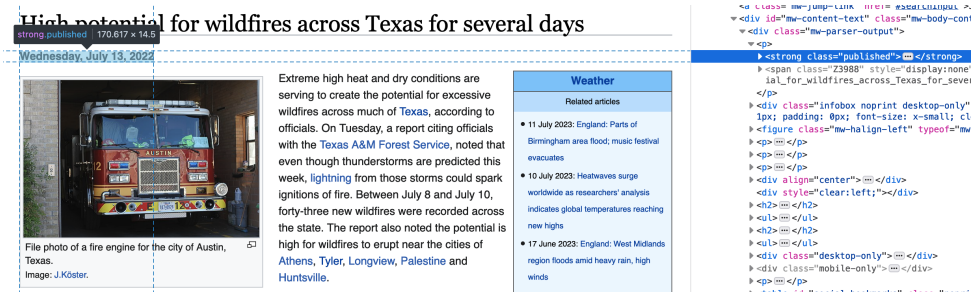

Figure 8: publish_date = soup.find("strong", class ="published")

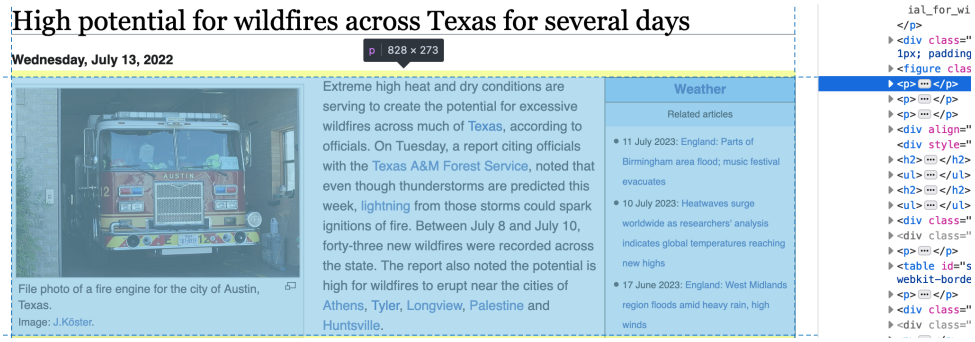

Figure 9: texts_content = article_content.find all('p', recursive=False)

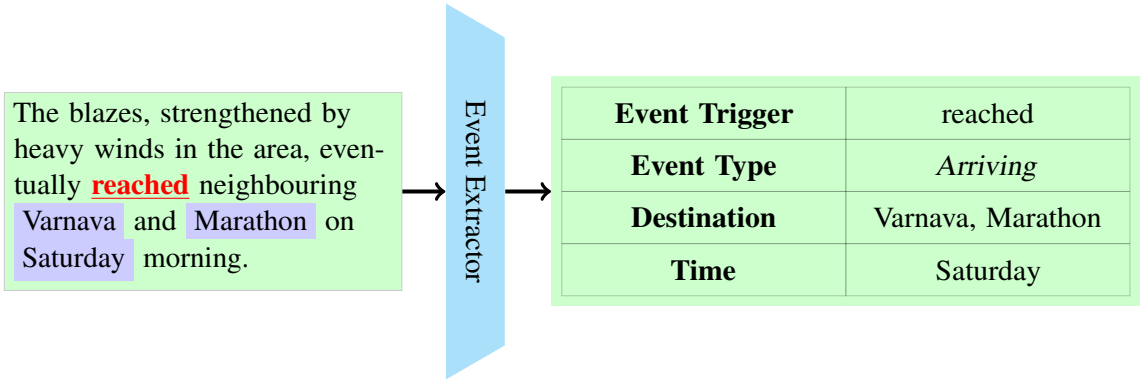

Figure 10: The structure of an event extracted from an input text.

Event	idTrigger	Trigger	EventType	idArgument	Argument	idRoleLabel	RoleLabel	Topic
	120	fire	type:catastrophe	Q35728	Trump	Q2394511	role:attacker	wildfire
	...	...	...	...	...	...	...	...

Table 1: Example of a relational database managed by a RDBMS.

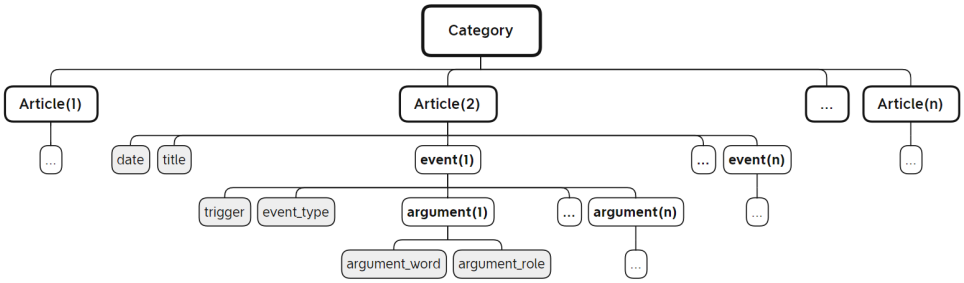

Figure 11: The tree structure of our XML documents.

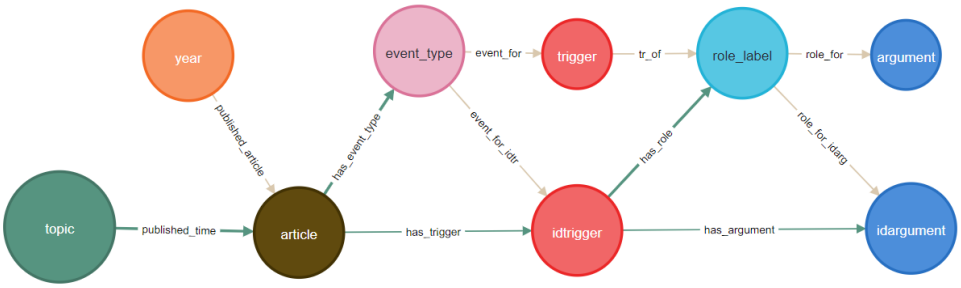

Figure 12: Graph structure in our database.

```
1 MATCH p=(:Topic )-[r1:published_time]→(arti:Article)
2 -[r2:has_trigger {text:"type:catastrophe"}]→(n:idTrigger)
3 where toInteger(right(r1.text,4)) >2020 return p
```

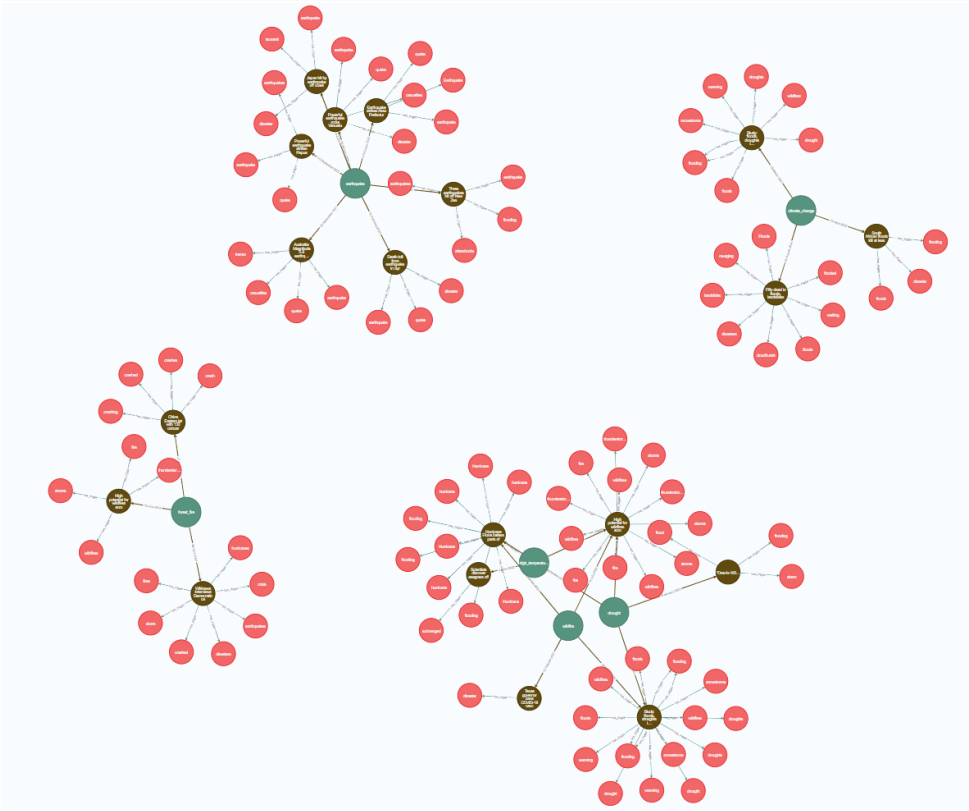

Figure 13: Query the distribution of triggers, articles and topics with event type *"catastrophe"* after 2020.

```
match (:idTrigger{text:"fire"})⟶(idarg:idArgument)
match p=(:Event_type)⟶(:Trigger{text:"fire"})⟶
(:Role_label)⟶(:Argument{text:idarg.text})
return p
```

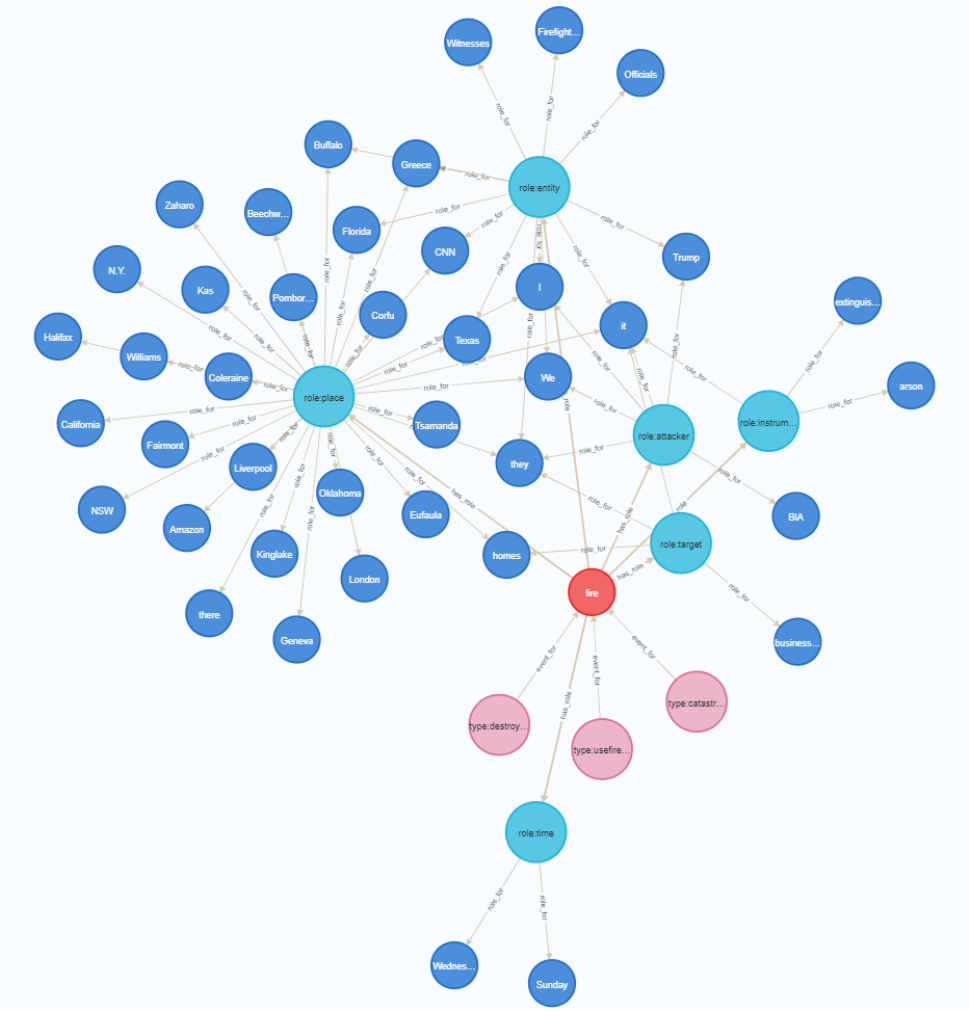

Figure 14: Visualize the distribution of event types, event arguments and roles of the trigger *"fire"*.

```
match  p=(t:Topic)⟶(arti:Article)-[et:has_trigger]→
(idtr:idTrigger{text:"fire"})-[rl:has_argument]→(:idArgument )
match (y:Year)⟶(arti)
return p,y
```

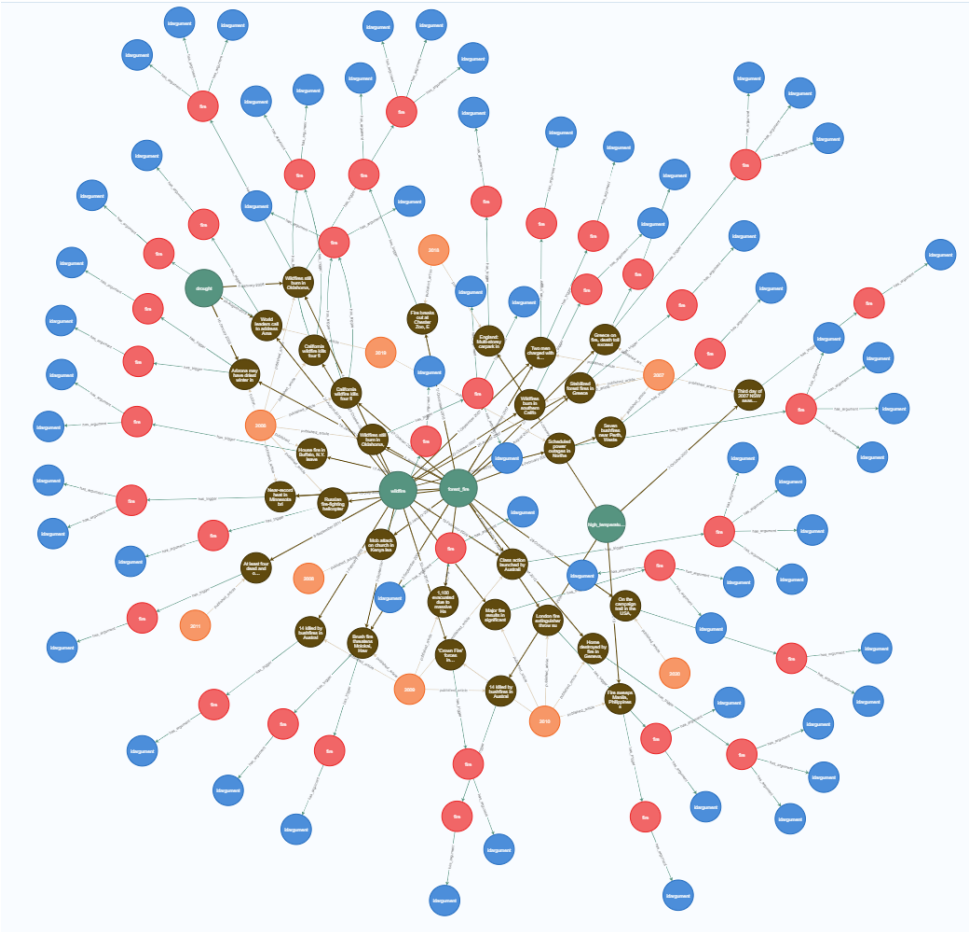

Figure 15: Visualize the distribution of the topics, articles, event arguments and publish years of the trigger *"fire"*.

```
match p=(topic:Topic)-[r1:published_time]→(arti:Article)⟶(idtr:idTrigger{text:"fire"})
match (topic)-[r1]→(arti)⟶(:Event_type)⟶(tr:Trigger{text:idtr.text})
match (year:Year)⟶(arti)
where toInteger(right(r1.text,4))>2003
with topic,year , count(tr) as num_tr
return topic.text as Topic,  toInteger(year.text) as Year, num_tr as num_trigger
order by Topic,Year desc
```

Table | Text | Code

Topic	Year	num_trigger
"drought"	2022	9
"drought"	2019	8
"drought"	2011	8
"drought"	2006	16
"forest_fire"	2022	1
"forest_fire"	2020	3
"forest_fire"	2018	1
"forest_fire"	2015	1
"forest_fire"	2011	8
"forest_fire"	2010	6

MAX COLUMN WIDTH:

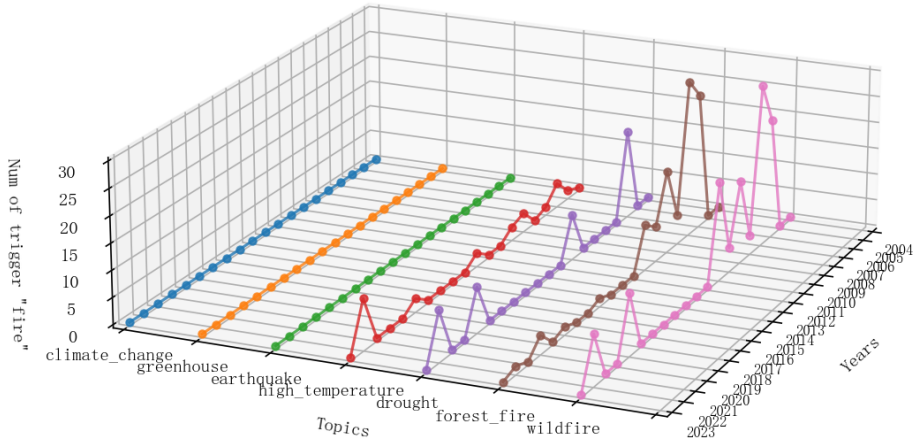

Figure 16: Count the frequency of the trigger *"fire"* under each topic after 2003.

```
match p=(topic:Topic)-[r1:published_time]→(arti:Article)⟶(et:Event_type)
where toInteger(right(r1.text,4))>2000
with topic,et,count(et) as num_et
order by num_et desc
with topic, collect(et.text)[0..10] as event_types, collect( num_et )[0..10] as num_event_types
return topic.text as topic, event_types, num_event_types
```

topic	event_types	num_event_types
"climate_change"	["type:statement", "type:causechangeofpositiononascale", "type:change", "type:communication", "type:agreeorrefus etoact", "type:causetobeincluded", "type:influence", "type:socialevent", "type:know", "type:processend"]	[98, 92, 89, 69, 52, 47, 47, 46, 46, 44]
"greenhouse"	["type:statement", "type:causechangeofpositiononascale", "type:communication", "type:change", "type:agreeorrefus etoact", "type:processstart", "type:socialevent", "type:know", "type:causation", "type:causetobeincluded"]	[88, 83, 58, 57, 44, 43, 43, 41, 40, 39]
"wildfire"	["type:statement", "type:catastrophe", "type:processstart", "type:bodilyharm", "type:motion", "type:reporting", "type:causation", "type:destroying", "type:causechangeofpositiononascale", "type:know"]	[88, 80, 62, 59, 59, 58, 58, 50, 43, 43]
"drought"	["type:statement", "type:catastrophe", "type:causechangeofpositiononascale", "type:reporting", "type:motion", "t ype:causation", "type:know", "type:influence", "type:processstart", "type:assistance"]	[83, 75, 60, 47, 44, 43, 42, 41, 38, 37]
"earthquake"	["type:catastrophe", "type:attack", "type:statement", "type:damaging", "type:reporting", "type:bodilyharm", "typ e:causation", "type:motion", "type:perceptionactive", "type:comingtobe"]	[80, 66, 63, 62, 61, 48, 48, 39, 36, 34]
"high_temperature"	["type:statement", "type:catastrophe", "type:causation", "type:causechangeofpositiononascale", "type:reporting", "type:bodilyharm", "type:motion", "type:creating", "type:know", "type:destroying"]	[78, 63, 53, 50, 49, 46, 43, 40, 40, 37]
"forest_fire"	["type:statement", "type:catastrophe", "type:destroying", "type:processstart", "type:motion", "type:bodilyharm", "type:causation", "type:reporting", "type:know", "type:arriving"]	[77, 68, 57, 51, 51, 51, 51, 43, 43, 39]

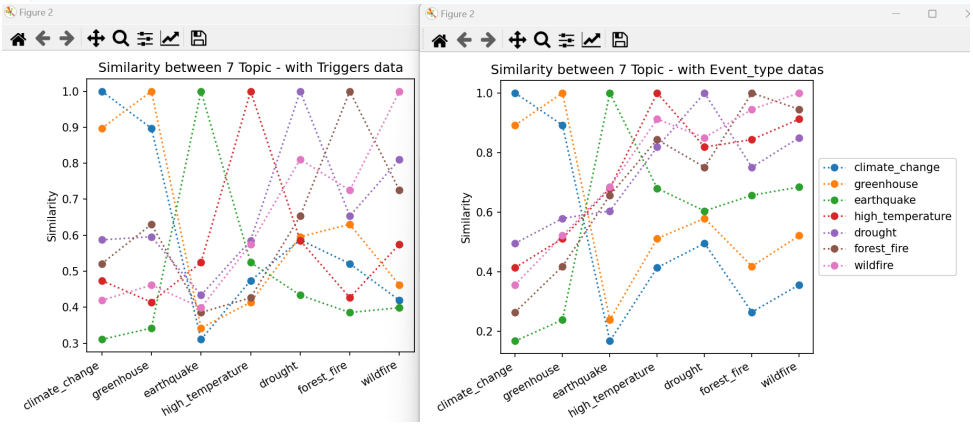

Figure 17: Count the first ten event types and triggers per topic to see the relationships between topics.