Paper DH06

No more merging! (property graph database demystified)

Alexey Kuznetsov, Grünenthal GmbH, Aachen, Germany

ABSTRACT

How many times have you heard that graph database is an excellent tool, but you could not imagine how these “bubbles” could help to make sense of your data? In this paper I will start with demonstrating “traversal” of clinical trial data using a small table-based self-written data visualization tool with a graph data model in the backend. A use-case is presented to demonstrate the advantages of using graph database for performing searches, looking at data from various perspectives as well as storing and retrieving different data sources and types in one place. The demo will be followed by describing basic principles of graph database and Cypher®. Technologies used: Neo4j® (graph database), Cypher(graph query language), Python® (+visualization package: plotly® dash).

INTRODUCTION

Graph databases help to find relationships between data and extract their true value. One of the most well-known graph databases is Neo4j, which is a service implemented in Java. For the purpose of this paper the GPL3-licensed community edition of Neo4j was used, which is available for readers to download and try at https://neo4j.com/ .
The author has extensive expertise in data handling using various programming languages and tools, such as SAS®, JMP®, SQL®, R®, Python, Pentaho® ETL, etc. He considers graph database in Neo4j’s implementation (together with complementary Cypher language and apoc library) to be one of the most efficient ways to manipulate data as well as to store data in a holistic, “whiteboard-like” way. The paper will focus on the data handling benefits of graph database, leaving other useful features provided with the database such as graph algorithms out of scope.
The author has developed a draft table-based data visualization tool with a graph data model in the backend, which allows a user to search data across all domains of a database, and to explore the data that is connected to that data.

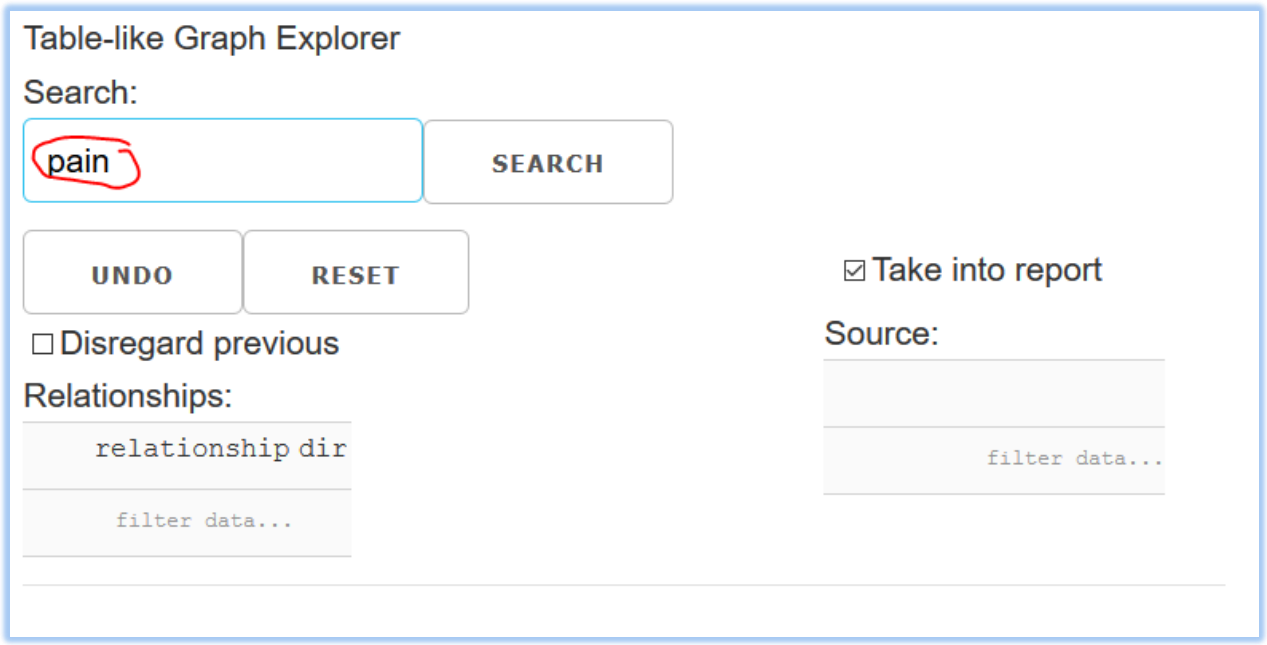
USE CASE AND DATA

For the use case, dummy computer-generated SDTM data of a clinical trial with 906 patients was used. (Data shipped with JMP Clinical® software).
In order to demonstrate the cross-domain/cross-datasource connectedness when using graph database, the demo includes data from SDTM domains DM (Demographics), AE (Adverse Events), CM (Concomitant Medications) as well as additional trial information/data from ClinicalTrials.gov. The use case can also be extended to other clinical domains as well as any other connected data including data from other industries.

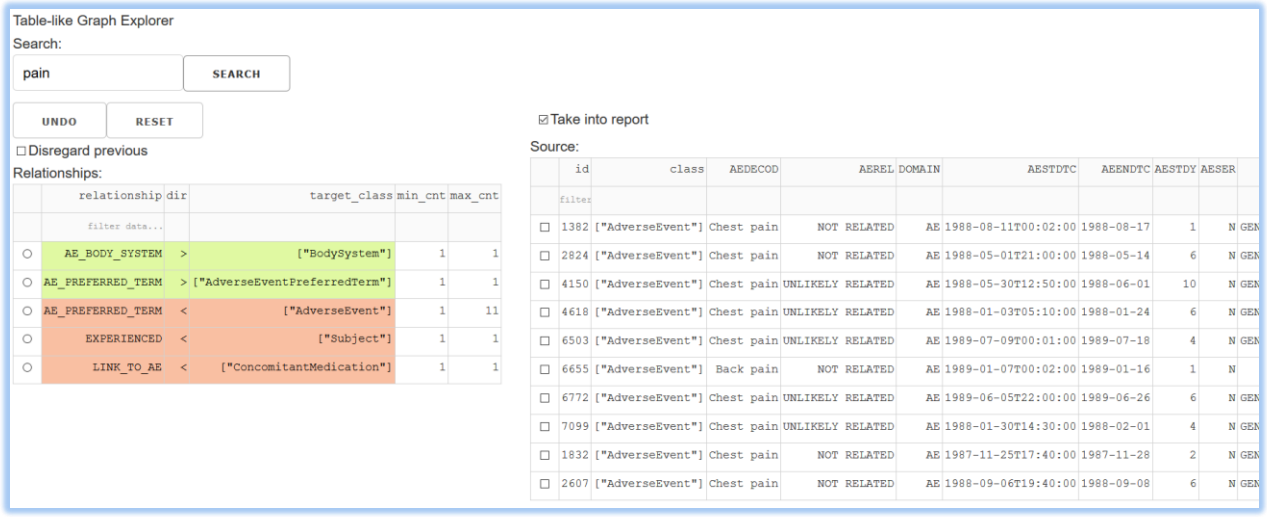
The paper will focus on data exploration. Loading and modeling of data in the graph database will be left out of scope.

USE CASE DEMO

Step1: Start with a cross-domain search in our graph database by entering “pain” in the search-string of the web application as an example:



Step2: After clicking on “Search”, 2 tables are populated on the page:



	relationship	dir	target_class	min_cnt	max_cnt
	filter data...				
○	AE_BODY_SYSTEM	>	["BodySystem"]	1	1
○	AE_PREFERRED_TERM	>	["AdverseEventPreferredTerm"]	1	1
○	AE_PREFERRED_TERM	<	["AdverseEvent"]	1	11
○	EXPERIENCED	<	["Subject"]	1	1
○	LINK_TO_AE	<	["ConcomitantMedication"]	1	1

	id	class	AEDECOD	AEREL	DOMAIN	AESTDTC	AEENDTC	AESTDY	AESER	
	filter									
☐	1382	["AdverseEvent"]	Chest pain	NOT RELATED	AE	1988-08-11T00:02:00	1988-08-17	1	N	GEN
☐	2824	["AdverseEvent"]	Chest pain	NOT RELATED	AE	1988-05-01T21:00:00	1988-05-14	6	N	GEN
☐	4150	["AdverseEvent"]	Chest pain	UNLIKELY RELATED	AE	1988-05-30T12:50:00	1988-06-01	10	N	GEN
☐	4618	["AdverseEvent"]	Chest pain	UNLIKELY RELATED	AE	1988-01-03T05:10:00	1988-01-24	6	N	GEN
☐	6503	["AdverseEvent"]	Chest pain	UNLIKELY RELATED	AE	1989-07-09T00:01:00	1989-07-18	4	N	GEN
☐	6655	["AdverseEvent"]	Back pain	NOT RELATED	AE	1989-01-07T00:02:00	1989-01-16	1	N	
☐	6772	["AdverseEvent"]	Chest pain	UNLIKELY RELATED	AE	1989-06-05T22:00:00	1989-06-26	6	N	GEN
☐	7099	["AdverseEvent"]	Chest pain	UNLIKELY RELATED	AE	1988-01-30T14:30:00	1988-02-01	4	N	GEN
☐	1832	["AdverseEvent"]	Chest pain	NOT RELATED	AE	1987-11-25T17:40:00	1987-11-28	2	N	GEN
☐	2607	["AdverseEvent"]	Chest pain	NOT RELATED	AE	1988-09-06T19:40:00	1988-09-08	6	N	GEN

“Source” table on the right with the actual data where term “pain” was found and “Relationships” table on the left which will be described in the next paragraph. “Source” table apart from the familiar SDTM columns contains a technical “id” column which represents the identifier of the corresponding node in the graph database (“node” can be considered an item of data, like a “row” in a table). It also includes a “class” column, which represents the class of data displayed. Classes are called labels in Neo4j, in CDISC terms they would be called domains.

The search term entered did not specify a domain to search in. The engine performed a search across all the domains (tables) and all columns and returned results in

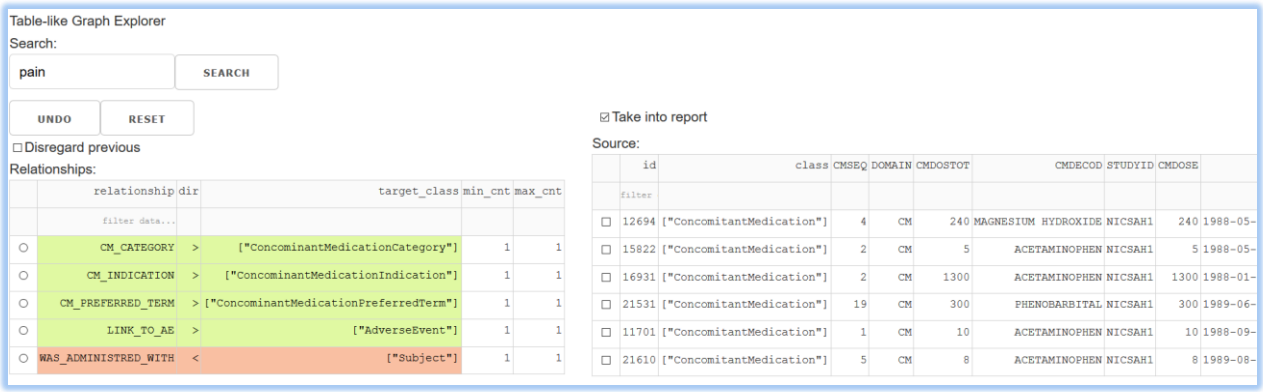
milliseconds. This is achieved by using full-text search functionality of Neo4j powered by the Apache Lucene® indexing which is provided out of the box. A link to documentation on how to setup full-text search in Neo4j is provided in the “Recommended reading” section below.

The “Relationships” table describes associations of the data in the “Source” to other data. In the column “relationship” you can find the type of the relationship, in the column “dir” – the direction of the relationship and in “target_class” – the class of data to/from which this relationship is routed. The best example of relationship in our case would be the relationship of type “EXPERIENCED” – i.e. we found adverse events experienced by subjects. Looking from the AE perspective the direction of the relationship would be to the left (“<”): in Neo4j terms: `(:AdverseEvent)<-[:EXPERIENCED]-(:Subject)` or flipped: `(:Subject)-[:EXPERIENCED]->(:AdverseEvent)`. In Neo4j Cypher query it does not matter in which order we put AdverseEvent and Subject as long as we specify the direction “<-“ or “->” correctly.

For better distinction the outgoing relationships (“>”) in the web interface are colored green and the incoming (“<”) are colored red.

The “Relationships” table serves as a means to further explore the data starting from the previous step. We can e.g. jump to the data about Body Systems of these adverse events, or to the data in Disposition Events (e.g. in case an AE led to drug withdrawal), or to the Subjects, who experienced these adverse events etc. The best example of the power a graph database provides from author’s perspective would be a jump to Concomitant Medications administered to treat these AEs: `(:ConcomitantMedication)-[:LINK_TO_AE]->(:AdverseEvent)`. Representation of this relationship in SDTM would usually require a linking table (RELREC).

Step3: By clicking on the circle to the left of “LINK_TO_AE”, the “Source” table on the right is repopulated with Concomitant Medication data of the CMs used to treat the AEs found in the first step. The “Relationships” table on the left is repopulated with the relationships routed to/from the CM data seen in the “Source” table.



It is important to notice that performance (speed) of such operations is very high, since there is no actual “merge”(“join”) that happens in the background. The data in a graph database is stored natively connected with the relationships you see in the table on the left. No “join” is required to perform the jump between two connected nodes, the AE node already “knows” the address of the related CM node.

Another advantage of a graph database is that there is no need to store many-to-many relationships in a linking table (like RELREC in SDTM). Each AE may have several relationships to CMs and a CM may have more than one relationship to AEs as well.

Step4: As a next step you can continue to “walk” through the database: e.g.

```
(:Subject)-[:WAS_ADMINISTERED_WITH]->(:ConcomitantMedication)
```

Table-like Graph Explorer

Search:

pain | SEARCH

UNDO | RESET

☐ Disregard previous

Relationships:

	relationship	dir	target_class	min_cnt	max_cnt
	filter data...				
○	CM_CATEGORY	>	["ConcominantMedicationCategory"]	1	1
○	CM_INDICATION	>	["ConcominantMedicationIndication"]	1	1
○	CM_PREFERRED_TERM	>	["ConcominantMedicationPreferredTerm"]	1	1
○	LINK_TO_AE	>	["AdverseEvent"]	1	1
○	WAS_ADMINISTRED_WITH	<	["Subject"]	1	1

Demographic characteristics of the subjects, who were administered with the previously identified CMs where received:

Table-like Graph Explorer

Search:

pain | SEARCH

UNDO | RESET

☐ Disregard previous

Relationships:

	relationship	dir	target_class	min_cnt	max_cnt
	filter data...				
○	DISPOSITION_EVENT	>	["DispositionEvent"]	4	4
○	EXPERIENCED	>	["AdverseEvent"]	4	8
○	IN_SITE	>	["Site"]	1	1
○	TREATED_WITH	>	["Treatment"]	1	1
○	WAS_ADMINISTRED_WITH	>	["ConcomitantMedication"]	9	38
○	HAS_SUBJECT	<	["Study"]	1	1

☑ Take into report

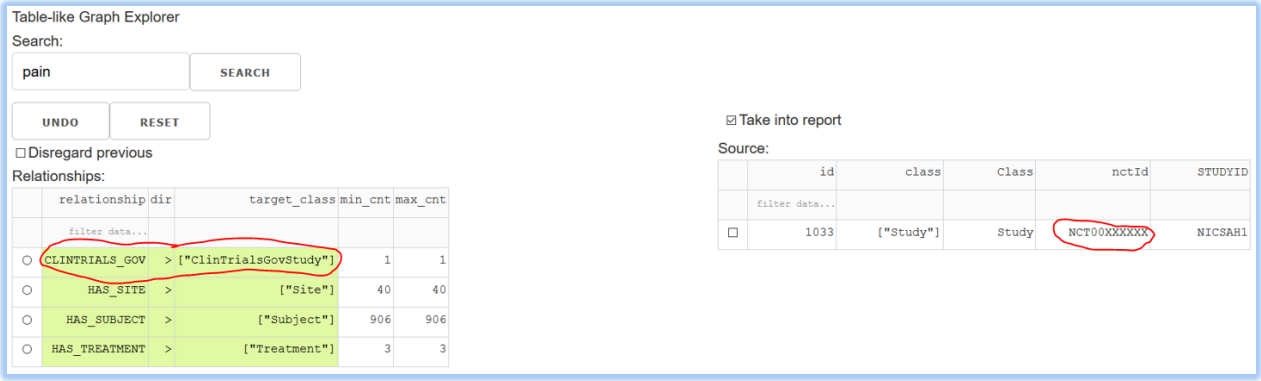
Source:

	id	class	DOMAIN	INVID	BRTHDTC	SEX	ARMCD	DMDTC	AGEU	SUBJID	SITEID	INVNAM
	filte											
☐	304	["Subject"]	DM	1A	1939-09-23	M	2	1988-04-25	YEARS	21008	02	021A
☐	491	["Subject"]	DM	2B	1945-01-11	F	2	1988-05-20	YEARS	282023	28	282B
☐	552	["Subject"]	DM	1A	1943-05-09	M	0	1987-12-25	YEARS	31003	03	031A
☐	857	["Subject"]	DM	1A	1945-11-04	F	0	1989-05-28	YEARS	461026	46	461A
☐	252	["Subject"]	DM	1A	1951-02-22	F	2	1988-08-30	YEARS	181014	18	181A
☐	860	["Subject"]	DM	1A	1949-05-11	F	2	1989-07-27	YEARS	461029	46	461A

Note, that there are 2 other columns in the “Relationships” table, which have not been discussed so far: “min_cnt” and “max_cnt”. They provide some summary statistics, that might be useful to get an overview of the data to expect after we perform the jump via a relationship. These numbers represent the number of nodes of the “target_class” data (i.e. out of the 6 subjects in “Source” table on the right, the minimal number of AEs observed per subject is 4, and maximum number is 8). These statistics are as well pretty “cheap” to calculate (in regards to performance) due to the design of graph database.

Step5: See what kind of data is available in the data model about the clinical study these subjects were part of. Next jump:

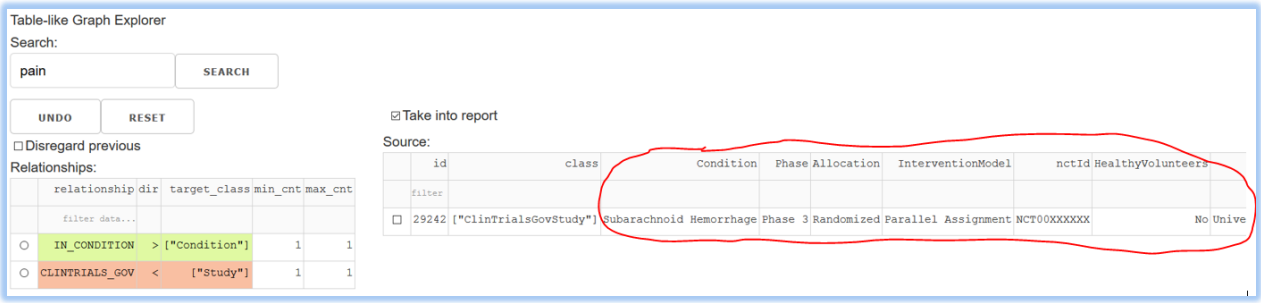
```
(:Study)-[:HAS_SUBJECT]->(:Subject)
```



Step6: When creating the data model in the graph database, dummy data were enriched with data from ClinicalTrials.gov: NCT00XXXXXX. After the next jump to the ClinTrialsGovStudy instance:

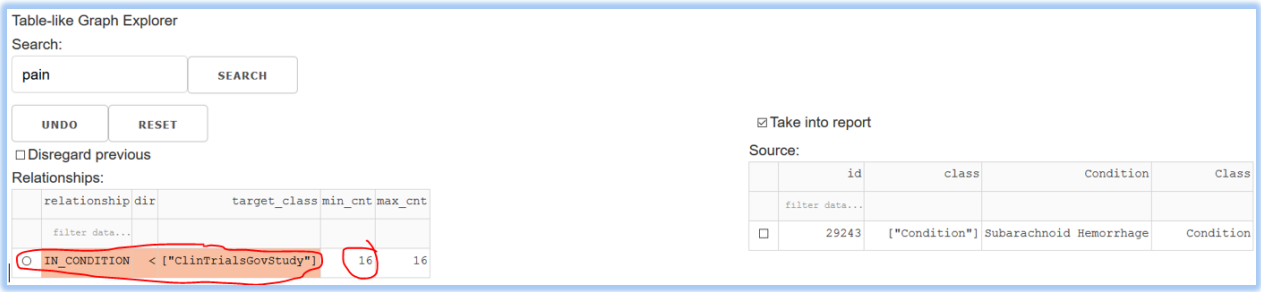
```
(:Study)-[:CLINTRIALS_GOV]->(:ClinTrialsGovStudy)
```

you can find the data typically included in the registry (e.g. Condition, Phase, Interventional Model etc.).



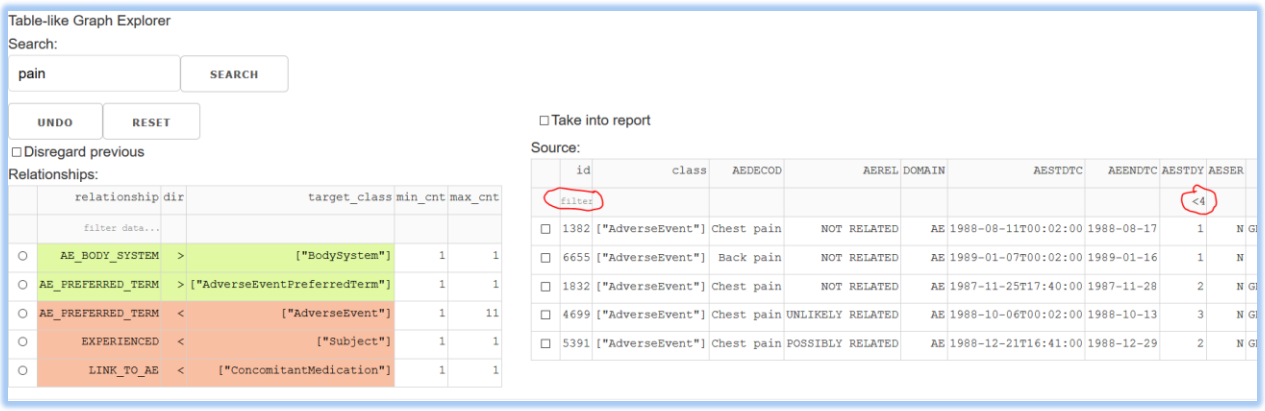
Step7: Depending on how you model the ClinicalTrials.gov data in the graph you can continue the journey. In this example, the “Condition” data from the ClinTrialsGovStudy node were extracted by the next jump:

```
(:ClinTrialsGovStudy)-[:IN_CONDITION]->(:Condition)
```



Step8: It would be possible to continue by taking a look at the other 16 nodes of ClinTrialsGovStudy in the “Subarachnoid Hemorrhage” condition available in the graph. From there, one could traverse further, if required …

The above examples gave a general idea about the graph data model and the advantages it provides. Please note that actually at each of the above steps filters could have been applied to the data: e.g. after finding the adverse events with key-word “pain”, a restriction to AEs that occurred before study day 4 would be possible (see following screenshot). This makes our data exploration possibilities even more flexible.



HOW DOES IT WORK IN THE BACKEND

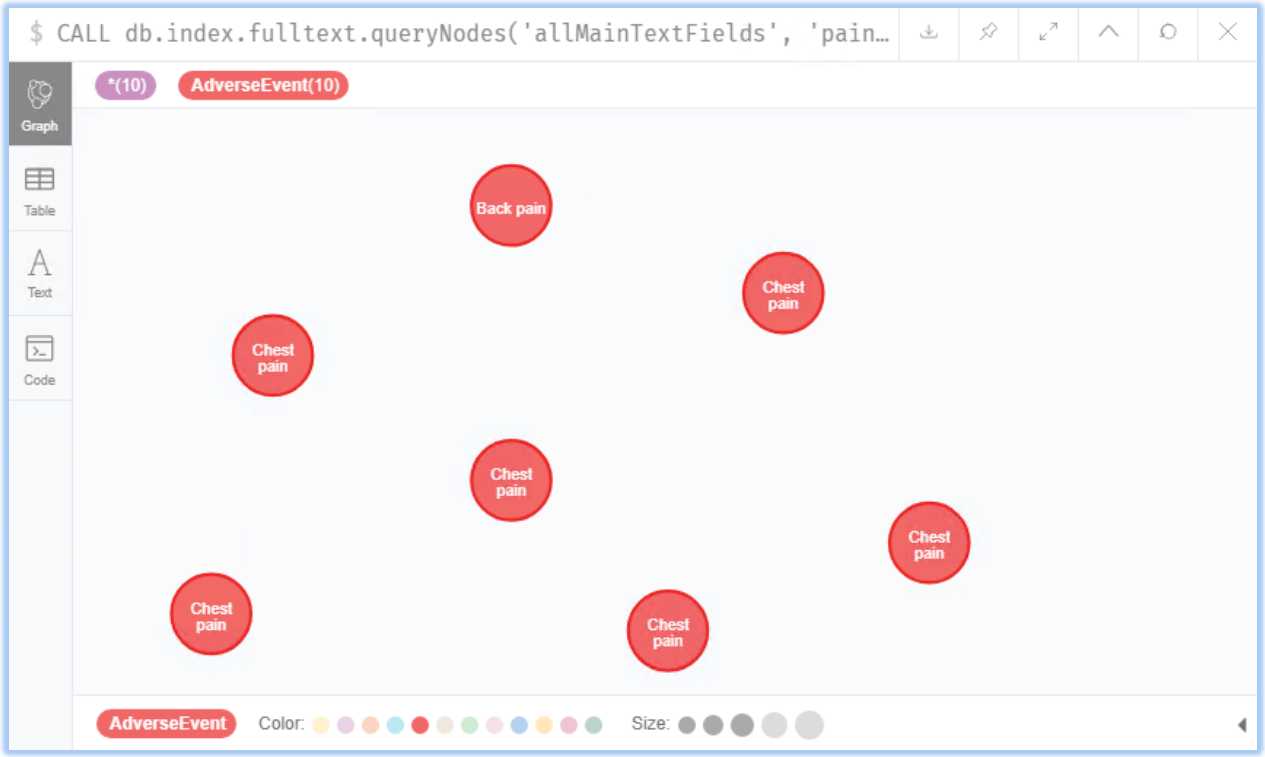
In this section it will be described what happens under the hood in the graph database – specifically the queries that retrieve the information reviewed in the previous chapter. The queries in Neo4j are written in “Cypher” language, which is a graph query language (similar to SQL for relational databases). “Cypher’s syntax provides a visual and logical way to match patterns of nodes and relationships in the graph. It is a declarative, SQL-inspired language for describing visual patterns in graphs using ASCII-Art syntax.” – citation from https://neo4j.com/developer/cypher-query-language/.

The first query utilizes the full-text search functionality provided by the graph database procedure. It finds all the nodes, containing words starting with “pain”:

```
CALL db.index.fulltext.queryNodes('allMainTextFields', 'pain*') YIELD node
return node
```

The node labels(classes) and properties in which the search is performed are predefined when setting-up the full-text index 'allMainTextFields' (for details see https://neo4j.com/developer/kb/fulltext-search-in-neo4j/)

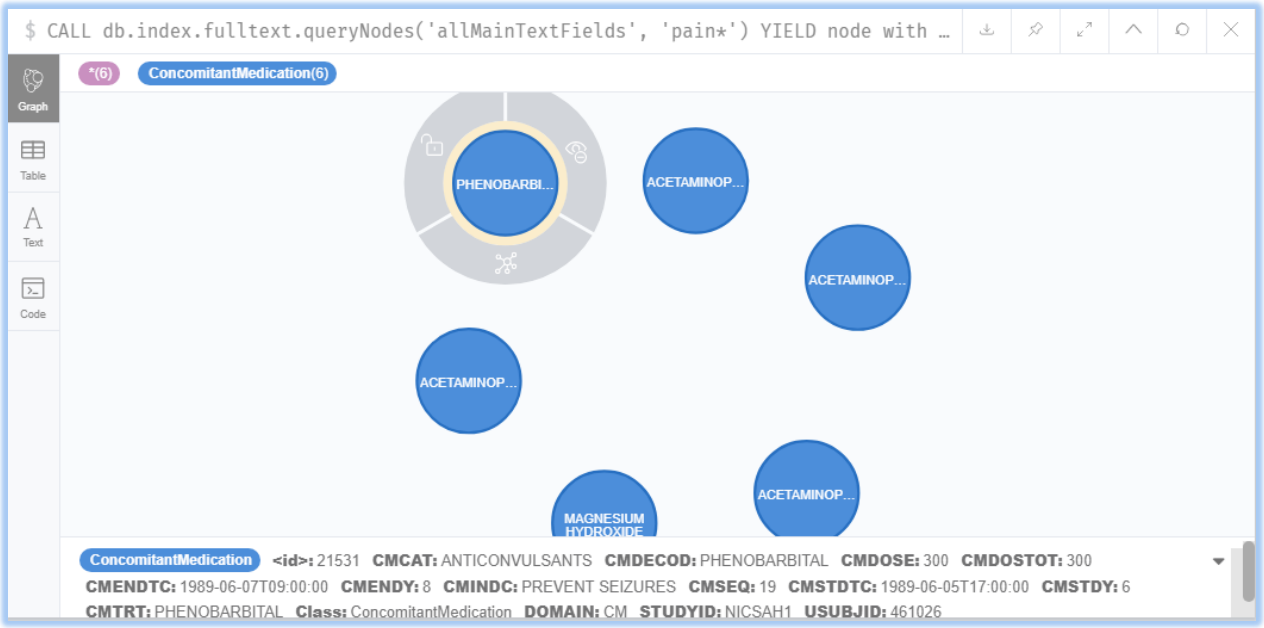
The result of the query in the Neo4j browser would look like this:



From here, after clicking on “LINK_TO_AE” in the web interface presented in the “USE CASE DEMO” section, the query modifies to

```
CALL db.index.fulltext.queryNodes('allMainTextFields', 'pain*') YIELD node
with node as ae
MATCH (ae)<-[:LINK_TO_AE]-(cm:ConcomitantMedication)
return cm
```

The result of the above query in the neo4j browser would look as follows:



Here, the nodes identified in the first step are named as “ae” and all the nodes that are labeled as ConcomitantMedication and connected to the ae nodes with a LINK_TO_AE relationships are matched. Afterwards the ConcomitantMedication nodes are returned. For future processing, the variable name “cm” is assigned to the newly identified nodes. At the bottom of the screenshot above you see the properties that actually sit inside the selected ConcomitantMedication bubble (CMDECOD is also displayed on the bubble itself). If the actual properties should be returned instead of visualizing the bubbles, the return statement would change to

```
return cm{.*}
```

producing an output in json format, that can later be visualized in the UI:

"cm"
{"CMSTDTC":"1988-05-01T06:00:00","CMTRT":"MAGNESIUM HYDROXIDE","CMDOSTOT":240,"USUBJID":"21008","DOMAIN":"CM","CMDOSE":240,"CMDECOD":"MAGNESIUM HYDROXIDE","STUDYID":"NICSAH1","CMENDTC":"1988-05-09T10:00:00","CMENDY":14,"CMINDC":"INDIGESTION PRN","CMCAT":"ANTACIDS/ADSORBENTS","Class":"ConcomitantMedication","CMSEQ":4,"CMSTDY":6}
{"CMSTDTC":"1988-05-30T04:00:00","CMTRT":"ACETAMINOPHEN","CMDOSTOT":5,"USUBJID":"282023","DOMAIN":"CM","CMDOSE":5,"CMDECOD":"ACETAMINOPHEN","STUDYID":"NICSAH1","CMENDTC":"1988-05-30T04:00:00","CMENDY":10,"CMINDC":"HEADACHE","CMCAT":"ANALGESICS/ANTIPYRETICS","Class":"ConcomitantMedication","CMSEQ":2,"CMSTDY":10}
…

In general Neo4j MATCH statements are followed by a structure like ()-[]->() where nodes are incapsulated in round brackets (), and relationships in square brackets []. If a type of relationship is specified, it is done inside these square brackets after a colon

```
()-[:LINK_TO_AE]->()
```

If the labels (classes) of the nodes are specified it is done inside the round brackets after a colon

```
(:ConcomitantMedication)-[:LINK_TO_AE]->(:AdverseEvent)
```

If the nodes of interest were already identified in the previous step of the query – there's no need to specify the label of those nodes again – just specify the name of the variable, used for those nodes in round brackets

```
(:ConcomitantMedication)-[:LINK_TO_AE]->(ae)
```

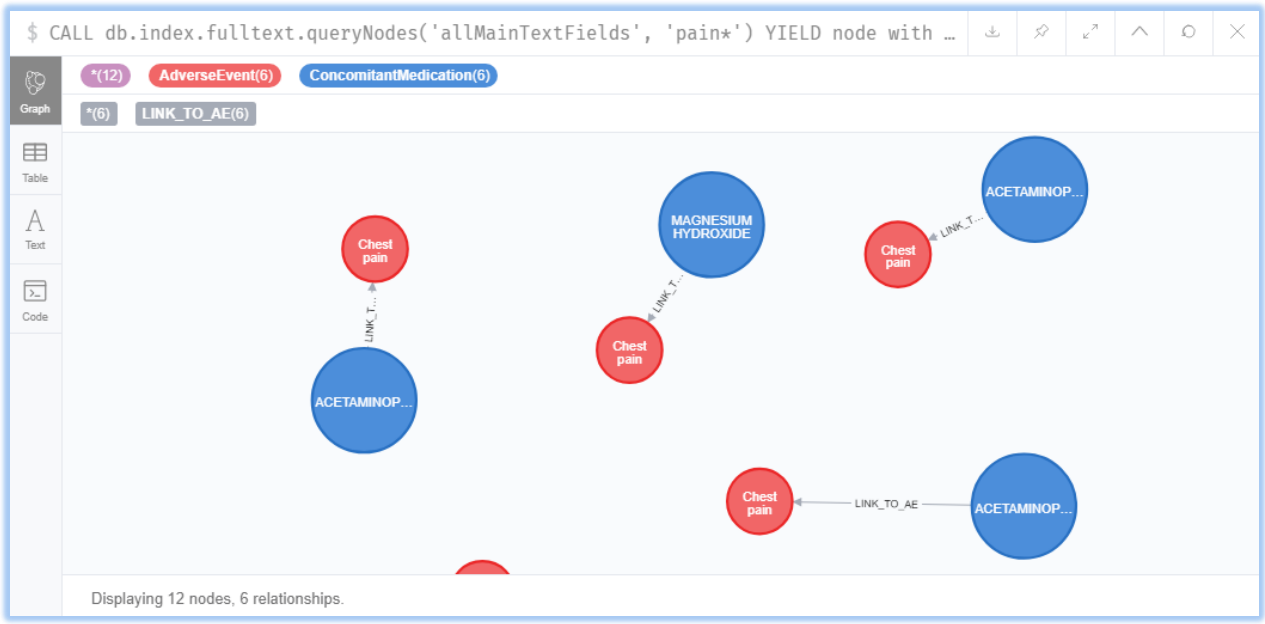
If some of the identified nodes need to be saved for further processing (or returning results), those nodes should be assigned a variable name – it is done inside of the round brackets before the colon:

```
(cm:ConcomitantMedication)-[:LINK_TO_AE]->(ae)
```

The complete path (connection) between the AEs and CMs can be returned by the following query:

```
CALL db.index.fulltext.queryNodes('allMainTextFields', 'pain*') YIELD node
with node as ae
MATCH p=(ae)<-[:LINK_TO_AE]-(cm:ConcomitantMedication)
return p
```

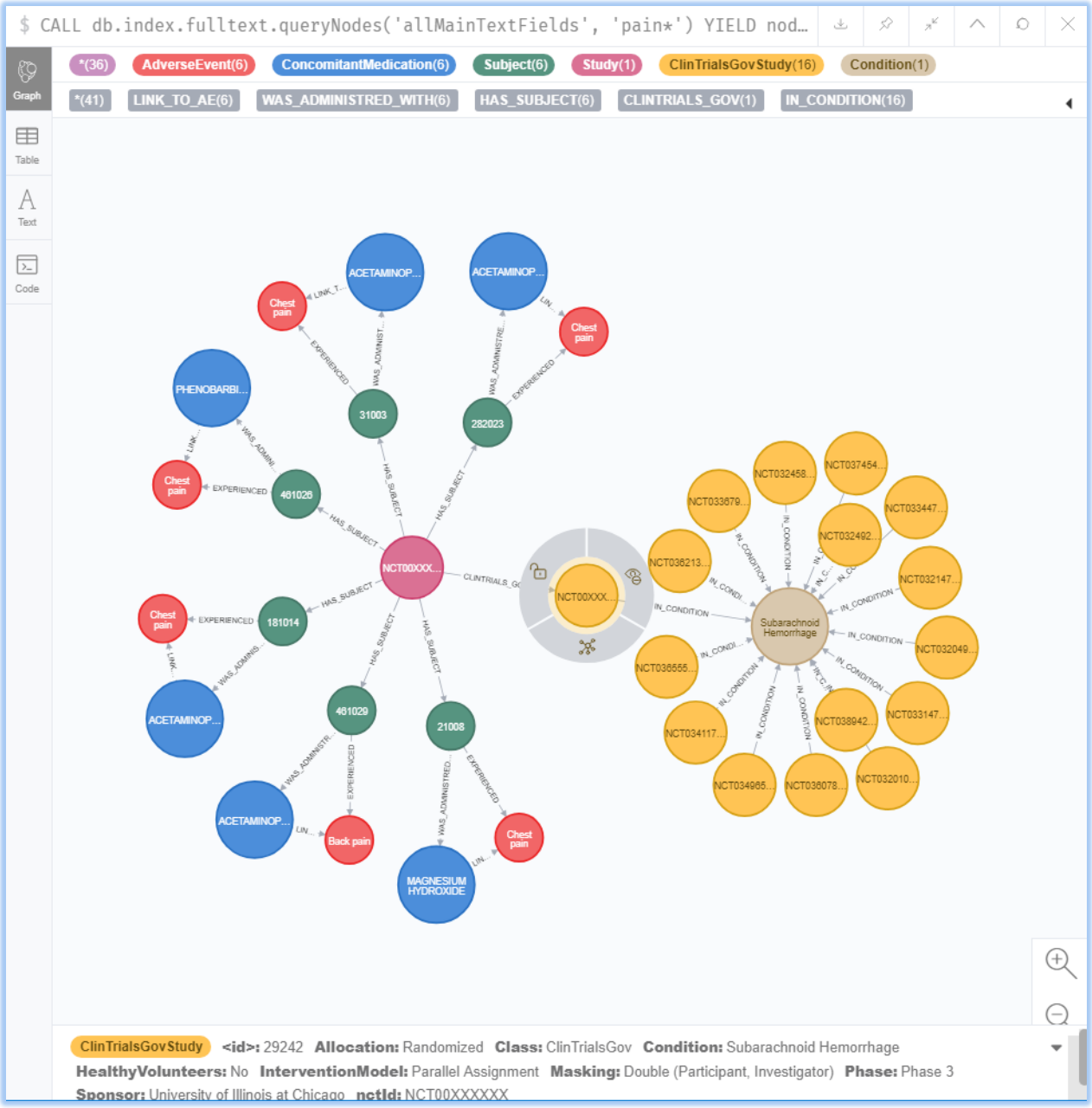
The result of the query in the neo4j browser would look like this:



Performing all the steps described in the previous chapter would lead us to the final query like this:

```
CALL db.index.fulltext.queryNodes('allMainTextFields', 'pain*') YIELD node
with node as ae
MATCH path=(ae)<-[:LINK_TO_AE]-(cm:ConcomitantMedication)<-
[:WAS_ADMINISTERED_WITH]-(subj:Subject)<-[:HAS_SUBJECT]-(study:Study)-
[:CLINTRIALS_GOV]->(ctg_study:ClinTrialsGovStudy)-[:IN_CONDITION]-
>(:Condition)<-[:IN_CONDITION]-(other_studies_same_cond:ClinTrialsGovStudy)
return path
```

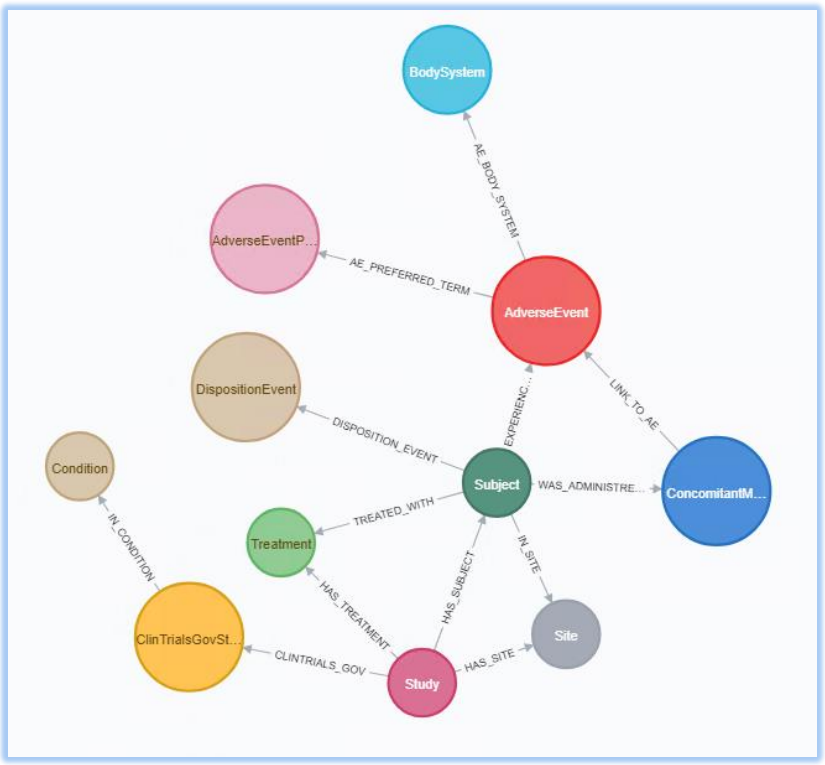
And the result of the query in the neo4j browser would look as follows:



Visualizations as above are very useful to demonstrate to users, how data domains are interconnected – especially to those users, that are not familiar with the data structure. In addition to custom query results (as above) it is possible to run the query

```
call db.schema()
```

to visualize how the data labels (classes) of the entire database are linked together:



For the purpose of web application demonstrated in the previous chapter, json outputs from neo4j were transformed to tabular format using python with pandas (python dataframe-handling package), and then visualized with added interactivity using python with dash (python visualization into web-app package).

CONCLUSIONS

In this paper, a practical example of clinical trial data exploration was illustrated using a web-application with a graph database in the backend and introducing basic concept of graph databases and the graph query language Cypher in parallel. The following conclusions are based on the experience during the application development and the data exploration:

- Graph databases can serve as a convenient backend for web applications
- Graph databases allow faster processing of queries on highly connected data compared to relational databases
- Graph databases allow storage of connectedness as data and eliminate the necessity to perform joins (merges) during queries
- Neo4j graph database provides useful libraries and indexes (including full-text search indexing) out of the box
- Cypher query language is intuitive and easy to learn
- Neo4j browser can be used to visualize a data model
- Web application development is possible with pure python without javascript knowledge
- The developed web application is pretty generic and would allow to walk through data of any domain after creating a graph model for the corresponding data

Unfortunately:

- As graph database technology is pretty new, it is hard to find experienced developers to support and/or further develop the code

- Plotly dash has limitations compared to pure javascript development and can be complex to learn and debug

Overall, graph databases seem to be a modern and powerful tool and a “must-have” in a data scientist’s/data engineer’s/data analyst’s toolbox.

ACKNOWLEDGMENTS

Many thanks to Sylvia Engelen (Data Science Lead Grünenthal GmbH), Heiko Hagedorn (Biostatistician Grünenthal GmbH) for reviewing the paper and providing useful comments.

RECOMMENDED READING

Intro to Graph Databases series:
https://www.youtube.com/watch?v=5Tl8WcaqZoc&list=PL9Hl4pk2FsvWM9GWaguRhlCQ-pa-ERd4U
Cypher query language:
https://neo4j.com/docs/cypher-manual/current/
Full-text search in neo4j:
https://neo4j.com/developer/kb/fulltext-search-in-neo4j/
Lucene search concepts:
https://lucene.apache.org/core/2_9_4/queryparsersyntax.html
Neo4j python driver:
https://neo4j.com/developer/python/
Python dataframe-handling package:
https://pandas.pydata.org/pandas-docs/stable/
Python visualization into web-app package:
https://dash.plot.ly/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Alexey Kuznetsov
Grünenthal GmbH
Zieglerstr. 6
Aachen Germany / 52078
Work Phone: +49-241-569-3424
Email: alexey.kuznetsov@grunenthal.com
Web: https://www.linkedin.com/in/alexey-kuznetsov-b67b8175