QL4POMR Interface as a Graph-Based Clinical Diagnosis Web Service

Sabah Mohammed, Jinan Fiaidhi(✉), and Darien Sawyer

Department of Computer Science, Lakehead University, Thunder Bay, ON, Canada
{mohammed,jfiaidhi,dsawyer}@lakeheadu.ca

Abstract. Most of the experienced physicians follow the SOAP note structure in documenting patient cases and their care journey. The SOAP note was originated from the problem-oriented medical record (POMR) developed nearly 60 years ago by Lawrence Weed, MD. However, the POMR/SOAP is not commonly found in electronic medical records (EMR) used today due to the flexible nature of building patient cases. Previous EMRs as well as clinical decision making software requires information to follow a strict schema which pushed POMR away from being implemented. However, the development of GraphQL is a wind of change that offer building a flexible interface and providing the query on the data that have semi structured schema like the POMR. This article developed a QL4POMR as a GraphQL implementation to the POMR SOAP note. Physicians can use this interface and create any patient case and present it for the purpose of diagnosis and prognosis with a varying backends. The QL4POMR implemented a mapping module to map graphs from POMR to HL7 FHIR and vise versa. Neo4j was used as backend to integrate all the data irrelevant of their nature as soon as the data is identified by objects and relations. The progress reported in this article is quite encouraging and advocates for further enhancements.

Keywords: POMR · SOAP · GraphQL · Neo4J · e-Diagnosis · Graph Databases

1 Introduction

Over the past five decades, healthcare over all the world is undergoing an immense transformation to tab on the technological advancement of having more pervasive connections with people and devices. The goals of these transformation include sensitive issues like promoting patient independence, improving patient outcomes, reducing risk, enforcing important policies like patient privacy, minimizing avoidable services, focus on prevention, understanding complex healthcare data, decrease medical errors, reduce cost, increase integration and partnership, clinical workflow consolidation and to timely prevent the spread of diseases. Connectivity in healthcare needs to go beyond the enabling connectivity infrastructure including wireless, mobile, cloud or any form of tele-health to include information connectivity. Much of the stated goals are only possible with highly connected information that goes beyond the hospital database silos. To make sense of the information connections and leveraging the connections within the healthcare existing data one need a holistic approach that uses graph representation and graph databases [1, 2]. Designing connected information based on graph databases produces efficient data management and data services at the same time [3]. In legacy healthcare systems, data services are often a missing component where data management is the only functionality that is only used for building error-tolerant and non-redundant database systems with traditional relational database query capability. Data services, however, provide retrieval, analytic and data mining queries capabilities encompassing high degree of relationships. There are other advantages behind supporting graph databases as they possess the ability to accommodate unstructured data and deal with schema less or to deal with structured data with strongly typed schema. In this research we are proposing a method to deal with all forms of data (with or without schema) as well as with data dealing with healthcare data having flexible schema that have defined upper structure but they vary with their subcomponents. In dealing with flexible schema, our method uses an interface for the popular GraphQL to build Web API for clinical diagnosis that uses a flexible schema like the POMR SOAP [4]. The GraphQL interface is connected to Neo4j schema less graph database. The GraphQL schema is build around the popular problem oriented medical record SOAP note that was introduced by Lawrence Weed [5] that describe medical cases based on the *subjective* observations as presented by the patient and the *objective* examinations conducted by the physician. Based on these two attributes the physician builds the *assessment* (i.e. diagnosis) and the *plan* to cure the patient case. SOAP is kind of diagnostic schema and a cognitive model that allows physicians to systematically approach a diagnostic problem by providing a structured scaffold for representing the clinical problem and all the associated clinical scenarios including physical exam, lab tests, assessments and planning. Based on SOAP schema physicians can reason about the chief complaint presented patient and identify the causes of presented encounters. By approaching SOAP, physicians can systematically access and explore individual illness scripts as potential diagnoses. SOAP is kind of clinical reasoning upper schema that uses a semi-structured model where the problem representation cannot be accommodated in a relational database model that requires a strict and well structured schema. SOAP can provide a well defined upper structure but the shape and contents of lower structures used depends on the purpose and the problem case. Clinicians highly support this approach as it provides several advantages including [6]:

- Flexible clinical case representation with lower problem structures that can easily be changed according to the differential diagnosis and the assessment progress
- Helps to manage physician cognitive load and maintain effective problem-solving. Flexible schema helps trigger clinicians to perform differentiating historical or physical exam maneuvers to refine the differential diagnosis.
- It help to teach others how to approach a clinical problem diagnosis
- It allow clinicians to adapt, refine and individualize their diagnosis schema by modifying, collapsing certain categories, or creating new ones, allows a schema to “work” best for them.
- Provide proper record keeping improves patient care and enhances communication between the provider and other parties: claims personnel, peer reviewers, case managers, attorneys, and other physicians or providers who may assume the care of the patients.

However, the major drawback of using semi-structured model is that the clinical data cannot be easily saved and retrieved as efficiently as in a more constrained structure, such as in the relational model. Records generated by the semi structured schema can only be accommodated using generic representation such as XML, JSON or OEM where clinical elements need to have unique ID or tag to enable their future reuse and retrieval. Based on these representations, clinicians can produce a range of different types of diagnosis problem cases, in efforts to meet different patient's encounters. To manage the data exchange for such semi-structured data, applications must follow the ISO/IEC 11179 Metadata Registry (MDR) standard[1] which paves the way to represent, share and query such semi structured data. The basic principle of data modeling in this standard is the combination of an object class and a characteristic. For example, the high-level concept "Chest Pain" is combined with the object class "Patient ID" to form the data element concept "Patient ID with Chest Pain". Note that "Patient ID with Chest Pain " is more specific than "Chest Pain". Efforts to implement the ISO 11179 in representing clinical cases resulted in using GraphQL API [7, 8]. In this article we are describing an interface based on GraphQL the POMR SOAP which follows the guidelines of the ISO 11179. The proposed interface is called QL4POMR which defines objects, fields, queries and mutation types. Entry points within the schema define the path through the graph to enable search functionalities, but also the exchange is promoted by mutation types, which allow creating, updating and deleting of metadata. QL4POMR is the foundation for the uniform interface, which is implemented in a modern web-based interface prototype. The QL4POMR interface is linked to the Neo4J Graph Database.

2 The POMER Flexible Diagnostic Schema

The problem oriented medical record (POMR) was introduced in 1968 as an attempt by Lawrence Weed (MD) to address the most common problems in diagnosis. It has been widely used afterwards by the medical community and approved by the American Institute of Medicine during 1974. Although there were no standard for it and no real implementation, the medical community from 2015 repeatedly restarted their efforts to implement it as POMR fits in the trend of care becoming more patient-centered [9]. A problem-oriented approach is also useful because patients and physicians can relate easily to the problems on the problem list and assess, update and respond to them [10]. Moreover, a problem oriented approach is seen as good solution to the bottleneck of interoperability by focusing on care services [11]. This design ideology been supported by the recent HL7 FHIR healthcare record representation based on service-oriented architecture for seamless information exchange. In this direction, POMR can be viewed as the upper uniform interface for defining every diagnostic problem. It will use a flexible schema where its top nodes are well defined and it lower level nodes vary according to the patient case. Figure 1 illustrate our view to the POMR schema where the yellow nodes (i.e. subjective, objective, assessment and plan) must exist while the remaining sub nodes marked green will vary according to the patient problem.

GraphQL is a query language for graph-structured data which is a common standard for querying semi structured data like the POMR. It has gained wide popularity with

[1] https://en.wikipedia.org/wiki/ISO/IEC_11179.

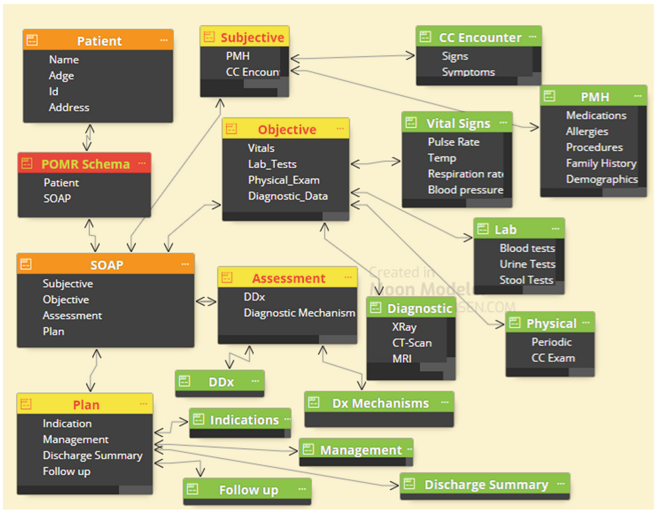


Fig. 1. The POMR flexible schema layout.

major IT venders like Facebook and Netflix [12]. Technically, GraphQL is a query language for APIs - not databases. It can be considered as abstraction layer providing a single API endpoint both for queries and mutations (i.e. data changes) for any database including NoSQL. The query objects are defined using GraphQL schema, which has an expressive expression to define objects, supports inheritance, interfaces, and custom types and attribute constraints. A GraphQL schema is created by supplying the root types of each type of operation, query and mutation (optional).

```
class GraphQLSchema {
  constructor(config: GraphQLSchemaConfig)
}
type GraphQLSchemaConfig = {
  query: GraphQLObjectType;
  mutation?: ?GraphQLObjectType;
}
```

The GraphQL server uses the defined schema to describe the shape of the semi-structured data graph. The defined schema describes a hierarchy of types with fields that are populated from the back-end data stores. The schema also specifies exactly which queries and mutations (changes) are available for clients to execute against the described data graph. GraphQL uses the Schema Definition Language(SDL) using buildSchema to define queries and the requested resolver. In the following example we are using the GraphQL SDL to fetch a Patient Object with name and age as attributes.

```
import { buildSchema } from 'graphql';
const typeDefs = buildSchema(`
   type Patient {
   name: String
   age: Int
 }
   type Query {
      getUser: User
   }
`);
const resolvers = {
   Query: {
      getUser: () => ({
            name: 'Sabah Mohammed',
            age: 66
         }
      ),
   },
};
```

GraphQL provides two important capabilities: building a type schema and serving queries against that type schema. So developers need first to build the GraphQL type schema which can be mapped to the required codebase. In this sense, GraphQL function is executing a GraphQL query against a schema which in itself already contains structure as well as behavior. The main role of GraphQL thus is to orchestrate the invocations of the resolver functions and package the response data according to the shape of the provided query. Such approach is generally known as *schema first programming*. In our case we design our POMR SOAP schema. GraphQL API provides set of tools (graphql-tools) a basic thin layer which include parse and buildASTSchema, GraphQLSchema, validate, execute and printSchema.

By using the graphql-tools we can connect the schema types with resolvers that may change its content.

```
const { makeExecutableSchema } = require('graphql-tools')
   const typeDefs = `
   type Query {
     user(id: ID!): Patient
     subjective: {
           type: Schema.Types.ObjectId,
           ref: 'Subjective'
       },
       objective: {
           type: Schema.Types.ObjectId,
           ref: 'Objective'
       },
       assessment: {
           type: Schema.Types.ObjectId,
           ref: 'Assessment'
       },
       plan: {
           type: Schema.Types.ObjectId,
           ref: 'Plan'
       }
   }
   type Patient {
     id: ID!
     name: String
   }`
   const resolvers = {
     Query: {
       user: (root, args, context, info) => {
         return fetchUserById(args.id)
       },
     },
   }
   const schema = makeExecutableSchema({
     typeDefs,
     resolvers,
})
```

Based on the schema design principles we can make our modification on top of what was reported by QL4MDR [7] to create our own QL4POMR schema that complies with the MDR ISO 11179. Figure 2 illustrates our overall QL4POMR echo system to define a diagnosis service interface that can work with the POMR semi structured schema and have all the nodes created to be stored at a graph database like Neo4j.

The QL4POMR is enforced with a CRUD Agent to manage the complexity of the schema and resolvers as well as to make structured querying through four modules (Create, Read, Update and Delete). For this we will need to use the *graphql-modules* library to facilitate building these modules. With using the graphql-modules, the CRUD Agent would be structured as follows:

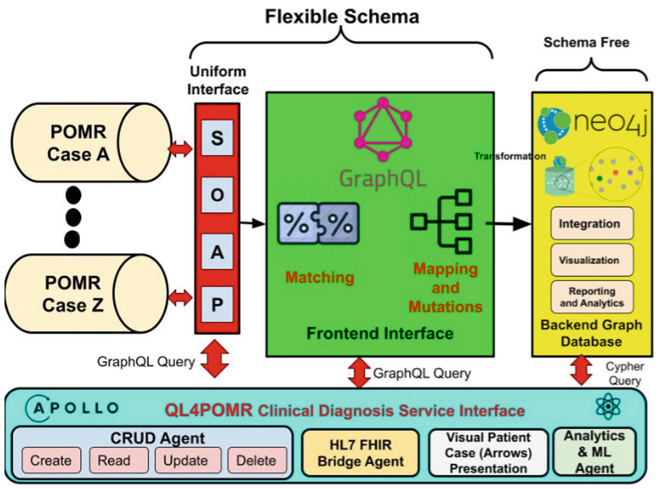
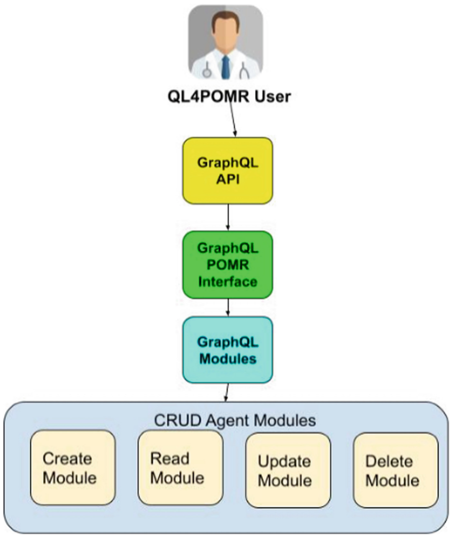


Fig. 2. QL4POMR diagnostic web service interface.



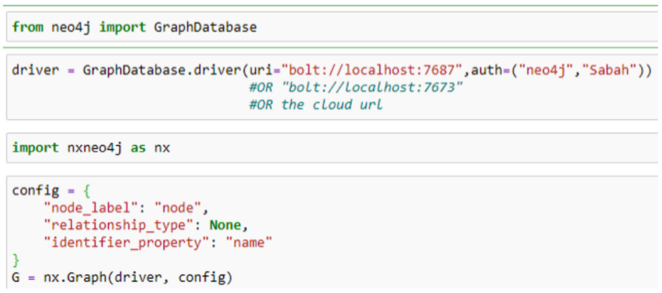
However, exchanging information between the QL4POMR interface and the Neo4j backend as well as with HL7 FHIR requires other primitive modules to manage matching and mapping operations required for facilitating information exchange. In the next two sections we are describing these primitive operations.

3 Interfacing QL4POMR with Neo4j

To connect QL4POMR to Neo4j we will need to use a programming language environment that have APIs for such connectivity. The best API for connectivity is the nxneo4j Python 3.8 library that enables you to use NetworkX type of commands to interact with

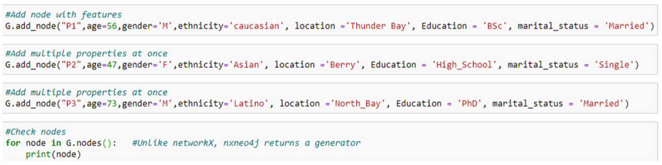
Neo4j. To demonstrate this connectivity we will use the following snippets from the Python Jupiter notebook:

(1) Connect to Neo4j



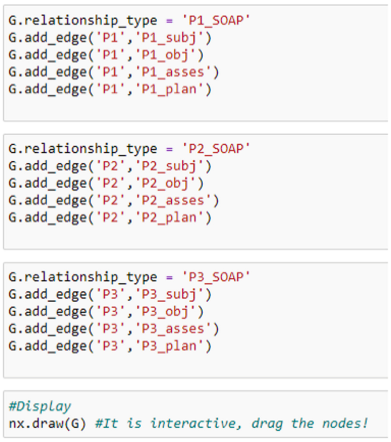
```
from neo4j import GraphDatabase

driver = GraphDatabase.driver(uri="bolt://localhost:7687",auth=("neo4j","Sabah"))
                              #OR "bolt://localhost:7673"
                              #OR the cloud url

import nxneo4j as nx

config = {
    "node_label": "node",
    "relationship_type": None,
    "identifier_property": "name"
}
G = nx.Graph(driver, config)
```

(2) Create some patient nodes:



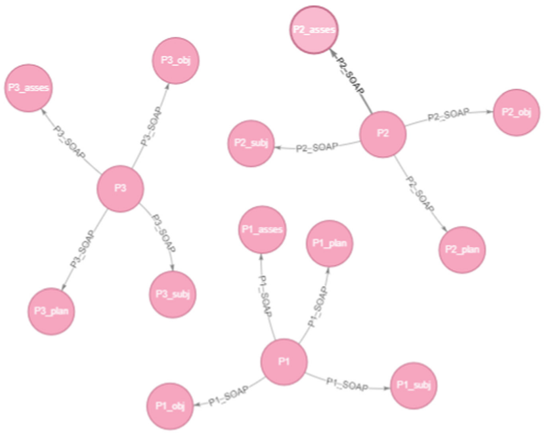
```
#Add node with features
G.add_node("P1",age=56,gender='M',ethnicity='caucasian', location ='Thunder Bay', Education = 'BSc', marital_status = 'Married')

#Add multiple properties at once
G.add_node("P2",age=47,gender='F',ethnicity='Asian', location ='Berry', Education = 'High_School', marital_status = 'Single')

#Add multiple properties at once
G.add_node("P3",age=73,gender='M',ethnicity='Latino', location ='North_Bay', Education = 'PhD', marital_status = 'Married')

#Check nodes
for node in G.nodes():   #Unlike networkX, nxneo4j returns a generator
    print(node)
```

(3) Create three POMR SOAP Nodes according to QL4POMR Schema:

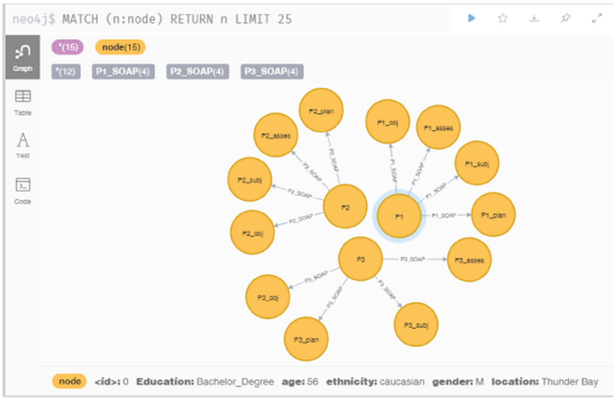


```
G.relationship_type = 'P1_SOAP'
G.add_edge('P1','P1_subj')
G.add_edge('P1','P1_obj')
G.add_edge('P1','P1_asses')
G.add_edge('P1','P1_plan')

G.relationship_type = 'P2_SOAP'
G.add_edge('P2','P2_subj')
G.add_edge('P2','P2_obj')
G.add_edge('P2','P2_asses')
G.add_edge('P2','P2_plan')

G.relationship_type = 'P3_SOAP'
G.add_edge('P3','P3_subj')
G.add_edge('P3','P3_obj')
G.add_edge('P3','P3_asses')
G.add_edge('P3','P3_plan')

#Display
nx.draw(G) #It is interactive, drag the nodes!
```

(4) Now we can see the three patient SOAPs displayed at the Jupiter notebook (Not Persistent see Fig. 3-a) and on the Neo4j (Persistent see Fig. 3-b)



(a)



```
neo4j$ MATCH (n:node) RETURN n LIMIT 25
```

node <id>: 0 Education: Bachelor_Degree age: 56 ethnicity: caucasian gender: M location: Thunder Bay

(b)

Fig. 3. Patient POMR patient data as displayed by jupiter (not persistent) and Neo4J desktop (persistent).

We followed this style of connectivity to build our Neo4j Graph database with POMR patient cases using our CRUD modules. However, the POMR patient data that resides at the Neo4j platform will provide clinicians with additional capabilities to monitor and update patient cases through the use of the Neo4J Cypher query language. For example executing the following Cypher query will provide information on the current patients encountered what conditions (see Fig. 4):

MATCH p = ()-[r:Encounter]- > () RETURN p LIMIT 25.

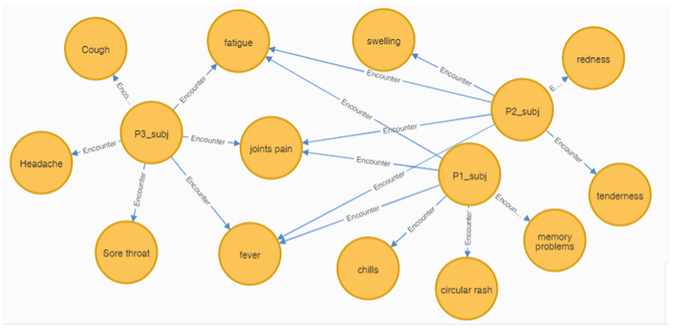


Fig. 4. Executing a Neo4j cypher query to browse what patients have encountered.

4 Interfacing QL4POMR with HL7 FHIR

Connecting with the HL7 FHIR requires mapping the QL4POMR types to the FHIR Resources types which are the common building blocks for all information exchanges with the FHIR. A single resource (e.g., Condition[2]) contains several Element Definitions (e.g., Encounter) which has Data Type (e.g. String) and cardinality associated with it (see Fig. 5). Figure 6 illustrates how the process of mapping the data types between the FHIR Resource and the QL4POMR. This is part of the matcher and mapper modules. Additionally we will need to use the graphql-fhir API[3] from Asymmetrik to connect to the FHIR server. Prior to the connection with FHIR server, we find that experimenting with the matcher and mapper based on FHIR sample record example is a useful practice. FHIR provide a JSON record example[4] for this purpose.

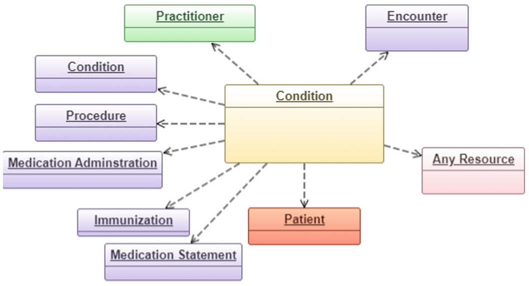


Fig. 5. FHIR condition data type.

[2] http://hl7.org/implement/standards/fhir/condition.html.
[3] https://github.com/Asymmetrik/graphql-fhir.
[4] https://www.hl7.org/fhir/patient-example.json.

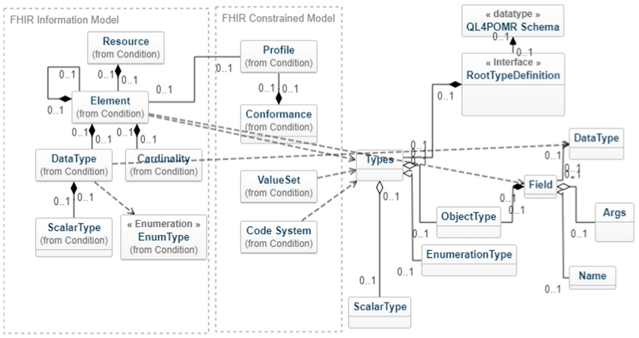

FHIR Information Model
Resource (from Condition)
Element (from Condition)
DataType (from Condition)
Cardinality (from Condition)
ScalarType (from Condition)
« Enumeration » EnumType (from Condition)
FHIR Constrained Model
Profile (from Condition)
Conformance (from Condition)
ValueSet (from Condition)
Code System (from Condition)
« datatype » QL4POMR Schema
« Interface » RootTypeDefinition
Types
DataType
Field
ObjectType
Args
EnumerationType
Name
ScalarType
0..1

Fig. 6. Mapping beteen FHIR resource and QL4POMR.

5 Connecting with Clinicians via Arrows

The QL4POMR interface allows the use of the Arrows.app[5] which is a new tool from Neo4j Labs. It will enable clinicians who have been trained on the POMR Types and naming convention to draw graphs for patient cases from using any web or mobile browser. The goal is not only to present the patient case but also to export its representation as JSON or Cypher file which can be integrated with the remaining Neo4J POMR cases. If the POMR naming were used correctly the matcher and mappers modules will be able to map them to the Neo4j graph database. Figure 7 illustrate how a physician can use the visual facilities of the Arrows app to describe a patient with ID34 who encountered joint pain and redness and his assessment with the skin test and blood test demonstrated a positive Cellulitis diagnosis and have been prescribed a Primsol as medication. Arrows can export the visual graph into several formats like Cypher which can be used to integrate the described care with the POMR patient cases. Figure 8 illustrate how the described case of the patient ID34 can be exported to Cypher. Once the Cypher file has been saved to the local storage, the QL4POMR interface will be able to use the matcher and mapper modules to integrate it with the Neo4J Graph Database. Having this visual tool will enable clinicians to focus of care design rather than training themselves to the information technology system that they are using. Figure 9 illustrate the patient ID34 case after mapping it to the Neo4J.

[5] https://github.com/neo4j-labs/arrows.app.

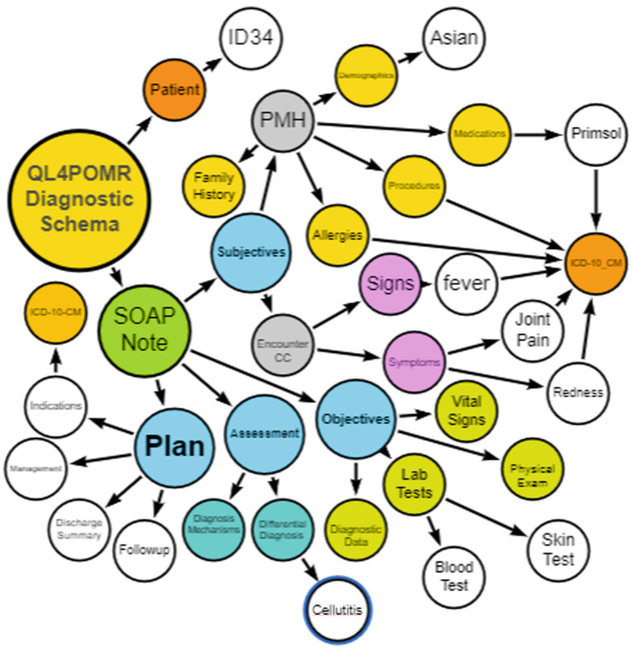
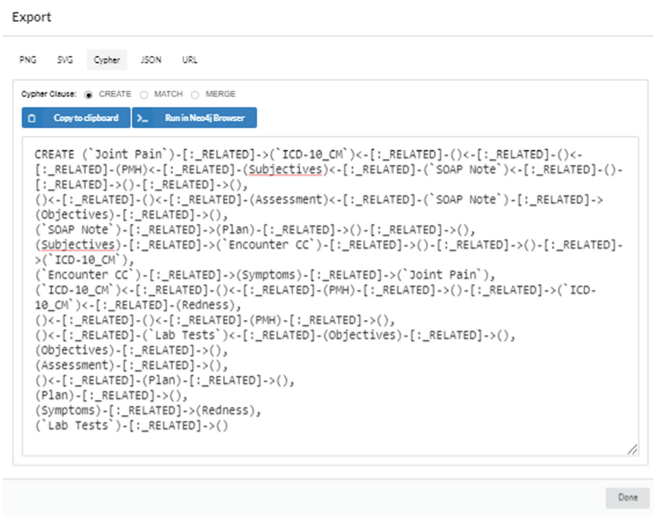


Fig. 7. Physician using arrows to describe a POMR patient case.



Cypher Clause: [✓ CREATE] MATCH MERGE

```
CREATE (`Joint Pain`)-[:_RELATED]->(`ICD-10_CM`)<-[:_RELATED]-()<-[:_RELATED]-()<-
[:_RELATED]-(PMH)<-[:_RELATED]-(Subjectives)<-[:_RELATED]-(`SOAP Note`)<-[:_RELATED]-()-
[:_RELATED]->()-[:_RELATED]->(),
()<-[:_RELATED]-()<-[:_RELATED]-(Assessment)<-[:_RELATED]-(`SOAP Note`)-[:_RELATED]->
(Objectives)-[:_RELATED]->(),
(`SOAP Note`)-[:_RELATED]->(Plan)-[:_RELATED]->()-[:_RELATED]->(),
(Subjectives)-[:_RELATED]->(`Encounter CC`)-[:_RELATED]->()-[:_RELATED]->()-[:_RELATED]-
>(`ICD-10_CM`),
(`Encounter CC`)-[:_RELATED]->(Symptoms)-[:_RELATED]->(`Joint Pain`),
(`ICD-10_CM`)<-[:_RELATED]-()<-[:_RELATED]-(PMH)-[:_RELATED]->()-[:_RELATED]->(`ICD-
10_CM`)<-[:_RELATED]-(Redness),
()<-[:_RELATED]-()<-[:_RELATED]-(PMH)-[:_RELATED]->(),
()<-[:_RELATED]-(`Lab Tests`)<-[:_RELATED]-(Objectives)-[:_RELATED]->(),
(Objectives)-[:_RELATED]->(),
(Assessment)-[:_RELATED]->(),
()<-[:_RELATED]-(Plan)-[:_RELATED]->(),
(Plan)-[:_RELATED]->(),
(Symptoms)-[:_RELATED]->(Redness),
(`Lab Tests`)-[:_RELATED]->()
```

Fig. 8. Exporting arrow visual graph into cypher.

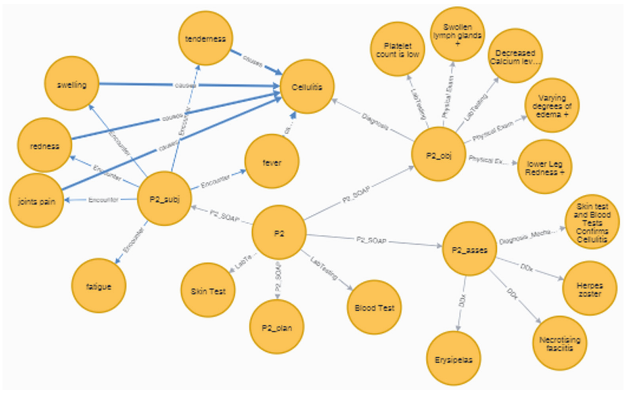


Fig. 9. Mapping a patient case from arrows to Neo4J.

6 Conclusions

Since the release of GraphQL as open source API for web development by Facebook in 2015 and the list of GraphQL users are increasing exponentially (e.g. Netflix, The New York Times, Airbnb, Atlassian, Coursera, NBC, GitHub, Shopify, and Starbucks) as this API enables the development of scalable systems which can deal with data interoperability. By adopting GraphQL developers can add new types and fields to the API, and similarly straightforward for the clients to begin using those fields. GraphQL's declarative model helps also to create a consistent, predictable API that we can use across all the clients. Developers can add, remove, and migrate back-end data stores; however, the API doesn't change from the client's perspective. Moreover, the declarative structure of the GraphQL allow developer to accommodate data of all sort including unstructured and semi structured data which a common case in healthcare applications. The framework includes a new graph query language whose semantics has been specified informally only to allow dealing with the unstructured nature of data and the heterogeneity of the sources. In this direction, GraphQL will fit designing clinical diagnostic services that collects all the patient information from heterogeneous data sources and inputs it into the system as well as to allow physicians to query and reason about it. This research paper illustrated how GraphQL can be used to implement the problem oriented medical record (POMR) and describing patient cases through is SOAP note. The QL4POMR is framework to systematically describe patient care journey and to integrate it with the other cases at the clinical practice. QL4POMR implemented a CRUD interface for describing the patient cases as well as to query the graph data via the Neo4J Cypher query. QL4POMR is also able to integrate with the HL7 FHIR healthcare record and map the POMR patient cases into FHIR record and vice versa. Our efforts to fully build the QL4POMR are supported by two national grants and we are anticipating more detailed to be published on this research project in the coming months.

Acknowledgment. This research is funded by the first author NSERC DDG-2021–00014 and first and second author MITACS Accelerates Grant IT22305 of 2021.

References

1. Kundu, G., Mukherjee, N., Mondal, S.: Building a graph database for storing heterogeneous healthcare data. In: Senjyu, T., Mahalle, P.N., Perumal, T., Joshi, A. (eds.) ICTIS 2020. SIST, vol. 196, pp. 193–201. Springer, Singapore (2021). https://doi.org/10.1007/978-981-15-7062-9_19
2. Singh, M., Kaur, K.: SQL2Neo: moving health-care data from relational to graph databases. In: 2015 IEEE International Advance Computing Conference (IACC), pp. 721–725. IEEE (2015)
3. Park, Y., Shankar, M., Park, B.H., Ghosh, J.: Graph databases for large-scale healthcare systems: a framework for efficient data management and data services. In: 2014 IEEE 30th International Conference on Data Engineering Workshops, pp. 12–19. IEEE (2014)
4. Mowery, D., Wiebe, J., Visweswaran, S., Harkema, H., Chapman, W.W.: Building an automated SOAP classifier for emergency department reports. J. Biomed. Inf. **45**(1), 71–81 (2012)
5. Cameron, S., Turtle-Song, I.: Learning to write case notes using the SOAP format. J. Counsel. Dev. **80**(3), 286–292 (2002)
6. Bayegan, E., Tu, S.: The helpful patient record system: problem oriented and knowledge based. In: Proceedings of the AMIA Symposium, p. 36. American Medical Informatics Association (2002)
7. Ulrich, H., et al.: QL 4 MDR: a GraphQL query language for ISO 11179-based metadata repositories. BMC Med. Inf. Decis. Mak. **19**(1), 1–7 (2019)
8. Ulrich, H., Kern, J., Kock-Schoppenhauer, A.-K., Lablans, M., Ingenerf, J.: Towards a federation of metadata repositories: addressing technical interoperability. In: GMDS, pp. 74–80 (2019)
9. Salmon, P., Rappaport, A., Bainbridge, M., Hayes, G., Williams, J.: Taking the problem oriented medical record forward. In: Proceedings of the AMIA Annual Fall Symposium, p. 463. American Medical Informatics Association (1996)
10. Simons, S.M.J., Cillessen, F.H.J.M., Hazelzet, J.A.: Determinants of a successful problem list to support the implementation of the problem-oriented medical record according to recent literature. BMC Med. Inf. Decis. Mak. **16**(1), 1–9 (2016)
11. Mukhiya, S.K., Rabbi, F., Pun, V.K.I., Rutle, A., Lamo, Y.: A GraphQL approach to healthcare information exchange with HL7 FHIR. Procedia Comput. Sci. **160**, 338–345 (2019)
12. Landeiro, M.I., Azevedo, I.: Analyzing GraphQL performance: a case study. In: Software Engineering for Agile Application Development, pp. 109–140. IGI Global (2020)