LION: LISTEN ONLINE. USING GRAPHQL AS A MEDIATOR FOR DATA INTEGRATION AND INGESTION

BY

DUSTYN JAMES TUBBS

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Kevin Chen-Chuan Chang

ABSTRACT

Data integration is the task of providing a unified view of multiple data sources. These sources can be, and are typically, heterogeneous in their data model, data query language (DQL), and data manipulation language (DML). In this thesis is described a system called "Listen Online", or Lion for short. Lion utilizes the GraphQL specification to provide integration for querying of web services. Lion provides a general structure by which arbitrary mediators can be used within a query. Lastly, by building on top of open source libraries, Lion provides the open source community with components that enable it to function in the form of GraphQL servers, visual layout libraries, and query builders.

To my lovely fiancee, soon to be wife, Marissa. You've shown me so much compassion and patience that I could not appreciate enough. Our futures will be brighter together.

ACKNOWLEDGMENTS

Nobody can do it all alone. I'm not an exception to that rule. I've had so many influences in my short time here that have helped me not only figure out what problem I'm working on, but how it fits within the grand scheme of everything. Firstly I would like to acknowledge my adviser, Dr. Kevin Chen-Chuan Chang whose focus on trying to solve a problem that people face resonated with me from our first meeting before I even started graduate school. Next I would like to acknowledge all the hardwork of the contributors of our "Social Universe" project: Zhaoheng Hu, Chen Wangl Lunan Li, Zongyi Wang, Naijing Zhang, and the countless people I've accosted with loaded questions like "hey, want to see something cool?". Lastly, I'd like to acknowledge the Department of Computer Science at the University of Illinois at Urbana Champgain. I never thought myself capable of being a small fish in a big pond but by Jove they did it and I am all the better of a student, scientist, and person because of it.

CONTENTS

Chapter 1: INTRODUCTION

Lion is a system that fits within the domain of data integration over a data model of web services. For sufficient background we need to look into both to get a rough idea as to the common models and problems being faced with each.

1.1 DATA INTEGRATION

Data integration is the problem of combining data residing at different sources, and providing the user with a unified view of this data [1].

Integrating between these different sources presents multiple challenges that an enterprising system designer needs to consider:

- Different data sources may have their own Data Querying Language (DQL) or Data Manipulation Language (DML). An integration system needs to provide a unified way to query these different sources
- Data sources may operate on fundamentally different models, such as relational, unstructured, array based, etc. There needs to be a method by which to map the logical relationships of entities to the unified view within the system.
- There is potential for missing data and irregular representation of the data. A data integration system should provide a mechanism by which these problems can be addressed such that the schema of that source can be consistent with the unified schema of the system.
- Data available at a source may exist at different levels of abstraction. For example, a database recording events may record them on a granularity of month/year and another database that is being integrated may record the data at a granularity of hours, minutes, seconds, in addition to day, month, and year. A data integration system should provide a mechanism by which the abstractions, too, are consistent with the unified schema.

These items do not span all of the considerations that one should have, but instead the chief considerations that can yield a system that has basic integration capabilities. These capabilities enable an important process for modern business analytics: the discovery of domain knowledge through data exploration and processing. This process, outlined in [2], follows:

1. Data are made available. These are either factual observations, or results from prior processing, and combinations thereof
2. Knowledge is made available. It derives from formal training and experience.
3. Knowledge about the data and its use is applied in two phases:
    3.1. Selection: subsets of available data are
        3.1.1. defined
        3.1.2. obtained, and
        3.1.3. merged
    3.2. Reduction: the data found are summarized to an appropriate level of abstraction
4. Several such results are made available and fused.
5. The combined information is utilized in two ways:
    5.1. Actions are taken that will affect the state of the world.
    5.2. Unexpected reusults will be used to augment the experience base of the participants, and others who receive this information, increasing their knowledge
6. The process loops are closed when:
    6.1. The actions and their effect are observed and recorded in some database.
    6.2. The knowledge is recorded to affect subsequent data definition, selection, or fusion.

An important distinction in this process is the difference between data and knowledge. Herein data is described as events of the real world that are captured either clerically or automatically. Knowledge, however, is the representation of abstract classes covering multiple instances of data and can be disproven using data.[3]. An example of this might be a database holding records of student enrollments. An individual record could represent a single enrollment for some student "Foobar McBazzington" in a database course such as CS 411, this would be data. Knowledge of these enrollments would involve providing some reduction on top of this data, such as the number of students enrolled being say 35 (an aggregation for counting). More particularly, some better knowledge might be that all students who enroll in CS 411 are at least juniors by credit hours (not by course requirement), Foobar McBazzington is enrolled in CS 411, therefore they must be at least a junior by

credit hours. This kind of knowledge is accumulated by domain experts (such as, in the case of this example, a registrar's office) and can be disproven by the data itself (if Foobar had a freshman level of credit hours as an example).

The accumulation of knowledge is the objective and the primary tool used for it are information systems that can integrate data. There is a rich variety of technique by which data integartion can be accomplished such as Logic techniques [4], Ontological techniques [5], or techniques that use artificial intelligence [6]. One of the more prevalant techniques that established the growth of the field [7] is that of mediators [2].

Mediators provide an abstraction between a source's data model and how it is accessed and represented within the information system typically by way of a schema. Some data models lend themselves very well to representation as a schema, such as fourth normal form of relational database systems [8], not all models are created equally and the models themselves historically evolve and intertwine their capabilities [9].

1.2 WEB SERVICES

The world wide web (WWW) exposes data in one of two ways. The first way is it can expose data in the form of HTML documents that are hosted by a web server. The second way it can expose data is through machine-to-machine interfaces that operate through some specification. We can describe the later as a web service.

1.2.1 REST

There are a couple interfaces by which these services operate. One of the more prevelant interfaces available presently is known as Representational State Transfter [10], or REST for short. The principals of this interface are built on top of the HTTP standard [11] to provide the following characteristics:

- A client-server model: to promote separation of concerns, a user interface is separated from the data concerns of the service. The API merely provides functionality.
- Stateless: To promote idempotency, a request in of itself needs to include all the data necessary to understand it. This promotes visibility for logging, reliability for recovering from failures, and scalability in that the web service doesn't need to manage resources to scale to N requesting clients.

- Cache-ability: A server should be able to mark data as being able to be cached. This lets a client cache determine whether or not perfoming another web request for data is necessary or if the client local data is sufficient.

- A uniform interface: REST-ful API's should adhere to four interface constraints. These are: identification of resources, manipulation of resources through representations, self descriptive messages, and hypermedia as the engine of application state. In practice these mean that data elements are described as resources, resource identifiers, representation, representation meta data, resource meta data, and control data.

- Code on demand: An optional constraint, a server can be able to expand the functionality of clients by way of providing code in the form of applets or scripts.

- Layered Architecture: In pursuit of the goals of a SOA stated above, a REST-ful architecture should restrict the knowledge of components within the architecture such that they can't see nor interact with services that they aren't immediately connected with.

Note that the constraints imposed by [10] **are not** a specification but an architectural style. The primary difference in this is that for an architectural style there can be a significant amount of variation with respect to the actual implementation. A specification, such as Simple Object Access Protocol (SOAP)[12], describes in exacting detail the exact nature of the data representation, communication protocols, and messaging. As a result we're faced with a "No True Scottsman" fallacy in that most implementations make some variation from the requirements of REST such that they become REST-like.

One thing that REST does not provide is a method by which to handle client authentication and authorization. Although the HTTP standard does have a simplistic method for this (carried over plain text), it's insufficient for keeping the computing resources of a web service secure. Whether in a Business to Business (B2B) environment or a end user application environment, REST-ful API's need to provide a mechanism by which authentication and authorization can be achieved. As of the writing of this document, there are a variety of identity protocols that can be used but the most prolific (in terms of those encountered during the development of LION) is Oauth2.0 [13].

Oauth2.0 specifies differing "grant" types by which applications can be allowed to access the web service:

- Authorization Code: public and confidential clients retrieve an authorization code from a user's login process to the web service and are provided an access token. These tokens

are typically short lived and can expire, although some services (such as Twitter and Reddit) provide tokens that do not expire.

- Implicit: A flow that enables special trusted clients to retrieve an authorization token without having to retrieve an authorization code.
- Password: A user's username and password are directly supplied to the web service and the application is returned an authorization token immediately. These applications are typically first party.
- Client Credentials: An application is able to use it's own identity to retrieve an access token outside the context of a user.
- Device Code: Applications accessing web services through the device code workflow typically have limited input capability and/or no browser. These devices can be found in specific hardware such as a Rouku or other streaming device.
- Refresh Token: A workflow that enables applications to utilize a refresh token (sometimes included with the authorization code workflow) for continued usage of a access token..

Within the work of Lion we primarily operate within the confines of the Authorization Code and Refresh Token work flows. In the context of Oauth2.0, Lion is a web application that is registered that enables users to log into a specific service. In terms of a machine-to-machine interface, REST doesn't provide much in terms of functionalities. If a web service exposes data and relations in a truly restful way (ie full URI's by which to access related resources), when a consuming application may have all the tools necessary to generate code/-markup by which to continue consumption of the service. In terms of really understanding what the service offers, however, REST does not necessarily expose the semantics of all operations available.

Instead, it's up to application designers to really understand the (sometimes unique and highly individual) work flows that a service may offer, as well as understanding the side effects of some operations. This information isn't exposed in a machine-to-machine interface but instead encoded within documentation. REST does presently control the market share so to speak when it comes to web services, but historically we did have a standard by which this kind of information was made available.

1.2.2 Service Oriented Architectures and the Semantic Web

This standard was in the form of SOAP as a Service Oriented Architecture (SOA). The goals of a SOA, pretty similar to that of REST, follows:

- Discoverable and dynamically bound: A service oriented architecture needs to be dynamically discoverable at runtime based on application criteria
- Self-contained and modular: A service oriented architecture needs to be sufficiently self-contained such that the services offered can be decomposed into sets of functionality, those can be compose to provide more complex interactions, the services can be easily understood within their domain, they are isolated from potential outages of services that they maybe composed of, and hides the details of its implementation
- Interoperability: A SOA needs to be able to communicate using a platform and programming language independent manner
- Loosly coupled: an architecture should have few-well known dependencies between the SOA's consumer and provider
- Self-Healing: A SOA should have the ability to recover from errors without human intervention.
- Stateless access: SOA's should provide idempotent methods of interaction.

These SOA's were typically described by a specification known as the Web Service Description Language (WSDL) [14]. This language provided a syntatic representation to describe, to a machine, the capabilities and logical entities that the web service operated over. In short, a WSDL provides enough information that a consuming service can understand the services, parameters they require, and the structure of the data that it returns.

Note, however, that this is purely a structural description of the web service offered through a SOA. What this description is missing is the semantics regarding the data available. The need to embed semantics with a SOA + WSDL was soon addressed by a concept known as the Semantic Web [15]. The primary idea behind this was that within the context of a web service, or web page, one can describe the granularity by which the data exists and operates as a taxonomy (classes of objects and relations between them) and a set of inference rules. An example of one such inference rule might be the following: cities have city codes, cities have websites, thus the websites can be associated with a city code (even if such association isn't encoded within the taxonomy or a database).

```
type Starship {
  id: ID!
  name: String!
  length(unit: String): Float
}

interface Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
}

type Droid implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  primaryFunction: String
}
```

```
type Human implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  starships: [Starship]
  totalCredits: Int
}

union SearchResult = Human | Droid
                           | Starship

enum Episode {NEWHOPE, EMPIRE, JEDI}

type Query {
  hero(episode: Episode!): Character
  droid(id: ID!): Droid
  node(id: ID!): SearchResult
}
```

Figure 1.1: Example GraphQL schema from `http://graphql.org/learn/schema/`

There have been multiple attempts to bring the Semantic Web as a reality by which semi-intelligent agents can operate over, but a glut of competing representations (such as through RDF [16] or OWL[17]) and lack of interest at the sheer work required to maintain the ontology[18] has made work stagnate. Ultimately the ideas that the Semantic Web strives for are really good, and still a good context by which to present research, but significant changes need to be made in order make it a more popular concept. The author suspects that proliferation of Microservices [19] may create a "what goes around comes around" situation with respect to web services where, after we recreate what a SOA represented utilizing modern interfaces, we also attempt to embed semantics as well.

1.2.3 Moving forward: GraphQL

Combining some concepts of a taxonomy with a modern interface, we have a new specification for web services called GraphQL [20]. Self-styled as a "query language for your API", GraphQL functions by a user developing a schema of entities and relationships. For example, Figure 1.1 describes a hypothetical schema on top of an API for Star Wars. Each

type by describing attributes as either scalars (a name in the case of the `Starship` type), or relationships to other types (`friends` in the case of a `Human` type).

Types can be defined by themselves, as an interface describing a shared set of attributes between multiple types, a Union of multiple types, or as an enumeration over predefined constants in the cases of the `Starship`, `Character`, `SearchResult`, and `Episode` types respectively with the `Human` and `Droid` implementing the `Character` type. The attributes of these types can be either a single value (e.g. the `name` of a `Character`), or a list of values (in the case of the `friends` attribute of a `Character`), distinguished by the brackets around the attribute type. Lastly, attributes may have arguments necessary to materialize an attribute. An example of this is the `length` attribute of a `Starship`. Here a string called the `unit` needs to be passed to the server to change the behavior of the response from the server.

```
query {
  hero {
    name
  }
  droid(id: "2000") {
    name
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    },
    "droid": {
      "name": "C-3PO"
    }
  }
}
```

Figure 1.2: An Example GraphQL query and result.

The interoperable method by which GraphQL handles requests for data and responses is that of queries in the GraphQL language. These queries use a JSON-like structure to describe traversals of relationships and acquire attributes. An example query and response from a GraphQL server is shown in Figure 1.2.

Chapter 2: RELATED WORK

Lion comes at an interesting time in the field of data integration. For contemporary and classic systems Lion does not come close to providing a comprehensive set of features that are typically found on these systems. If one were to examine the features offered by Lion, in the context of data integration, it would be focused primarily on querying for data sources.

These systems still describe some of the important ideas within the field of data integration that also helps place what Lion provides to the field. We will discuss what Lion brings to the table relative to these systems soon. For now, we take a quick delve into some prolific data integration systems as well as systems that perform integration over the web.

2.1 DATA INTEGRATION SYSTEMS AND IDEAS

Data integration systems commonly use a technique known as mediators. Mediators have been defined canonically by [2] as "a software module that exploits encoded knowledge about some sets or subsets of data to create information for a higher level of applications". Here the definition of data may or may not necessarily be representative of the underlying values that are stored in a database, but instead facts by which an application layer operates. This means that one may have a mediator for a specific kind of database to provide a unified interface to access the data, typically called a wrapper, or mediators that provide additional functionality on top of other mediators. This functionality can be the mapping of abstractions between sources, the indexing of different mediators (called meta-mediators by [2]), or any other kind of functionality that is desired. Note that, as defined, these mediators **need** to be small in their scope and maintainable by one expert or at most a small group of experts [2].

There have been a variety of systems that have been developed using the model, but we'll be looking toward a small subset of these systems. Namely, we'll be having a quick look at the use of data mediators within the TSIMMIS project [21], Garlic [22], and lastly the BigDawg poly store system [23]. This is not necessarily a comprehensive list but two of the most impactful historic systems and one of the more recent systems.

2.1.1 The TSIMMIS Project

In the TSIMMIS project [21], the authors distinguish a system for data integration of heterogenous data sources. This system was an influential precursor for future systems,

working on an underlying model that, looking back, was very extremely thoughtful for future data base systems. Namely, they assumed that the underlying data was varied in it's structure (ie unstructured, semi-structured, or structured), that there may not be a regular schema to describe the data, the meaning of the data may change and may be incomplete. To solve this issue the authors utilized the Rufus project [24] to present an object oriented representation of the underlying data.

Within the TSIMMIS architecture this provided values to an information source. Information sources themselves has a translator that take queries made within TSIMMIS' *Object Exchange Model* query language and returns objects in the form of tuples. TSIMMIS also provides a user interface to run queries and view the resulting data objects. TSIMMIS's model works by providing a method of generation for wrappers and mediators, which is outlined in [25]. Overall this project's contributions are immense and still felt in contemporary works.

2.1.2 The Garlic Approach

Garlic [22], conversely, is a system doesn't just operate on semi-structured text, but instead, multi-media information. The Garlic approach with respect to data integration isn't to necessarily provide all the functionality of a data integration system through meidation, but instead provide a unified schema by which other mediators/systems can operate on. This unified schema follows an object an object-oriented data model standard outlined in [26]. By working on top of this abstraction, Garlic provides the following functionality: an object-oriented extension of SQL that enables clients to access data repository wrappers and a runtime system that enables client applications to connect to Garlic and interact with the objects within the repositories.

Essentially, Garlic provides a means by which underlying wrappers can be represented within a data mediation system without being heavily integrated. In [21], the authors heavily integrate the mediation and wrapping of a data source as a single entity. Conversely, Garlic provides an interface by which wrappers can be developed against, effectively making a looser coupling but coming at a cost of losing the generative capabilities as described in [**?**]. This isn't necessarily without some benefit as, outlined in [27], the wrappers themselves can have more involvement in query planning for the purpose of optimization.

2.1.3 Big Dawg

Lastly, we have a quick look toward the BigDawg polystore system [23]. A more contemporary system, BIgDawg approaches the problem of data integration from the perspective of relative transparency to the underling data model. Data sources connect to logical "islands" by way of a "shim" which can be thought of as a mediator that converts the island's language to the language of that particular source. An example would be a "Relational" island which may have a shim to say PostGres. The islands help manage query optimization and concatenation of data values, but does little else in terms of computation.

For queries that span multiple islands (ie a relational island to a plain text island), a CAST capability is supplied to a user. This process allows one to have a query that yields a data type (say array data type) and use the values from that island within the query language of another island (say a relational type). This process, however, is not automatic and the user must specify it themselves. BigDawg's contributions within data integration and mediation is not necessarily to provide a unified schema by which users/programs can operate on, but instead work toward providing the full functionality of the underlying source as well as data transparency as to where the values are specifically within the data store.

The BigDawg approach is similar to that of Garlic and TSIMMIS insofar as we want to present a user with a uniform way of accessing the data. The latter two believed that unified schema that follows an object oriented strategy was the path to success, whereas the latter thought that the approach to be fully transparent to the underlying data model was a requirement. This observation can primarily be made by considering that BigDawg's work takes place after approximately 20 years of database development. Within that time frame we've seen far more in terms of sources available that Garlic and TSIMMIS could consider. This includes the NoSQL movement for document stores such as MongoDB [28], array databases for Science data management like SciDB [29], streaming databases like Gigascope [30], and any other kind of implementation. In essence, BigDawg doesn't attempt to unify a schema representation of the underlying source but instead provide querying capabilities on top of these heterogenous database systems.

2.2 THE WEB AND DATA MEDIATION

Data integration and mediation within the context of web services and web accessible documents presents its own sets of interesting challenges. In the case of documents we have a mixed bag of pages that may or may not follow standards set by the Internet Engineering Task Force (IETF). In the case of services, we may or may not have services that disclose data

that programmatically describes what data is available at the service and how to retrieve it. If the web service in particular follows a service oriented architecture then it should disclose a programmatic description.

Prior art explores a small variety of techniques, here we focus on the use of wrappers and the Semantic web. The use of wrappers is one that comes naturally when viewing it in the context of the prior art of data integration. The systems we've discussed thus far typically make use of mediators in the form of wrappers or translators that intake a system specific data query language or data manipulation language to access the data. For accessing web pages we are immediately presented with a data querying language in the form of accessing a page and parsing its contents. The authors of [31] explore this line of thought in the context of data integration by providing a method to semi-automatically synthesize a wrapper.

Their train of thought with respect to web pages is that those that can function sufficiently as a data source typically use regular structure with respect to the data. That is, the values available on a page are typically within a hierarchy of the documents structure and described by styling tags. The authors utilize the behavior of LEX and YACC [32] to create a grammar for the logical entities found on the page. This grammar specification is used to materialize the data within an integration system.

In their work they also provide a good view as to the state of the art (at the time) of wrapper generation with respect to a web page. The evolution of the web in respect to the availability of data has shifted from being available in directory-style web pages accessible strictly through a user's web browser to that of web services. As such, work of data integration for the web, too, has shifted to encompass both web pages and services themselves. We are going to look into an approach that provides semantics on top of a web service.

The authors of [33] approach data integration for web services on top of these two assumptions. By doing so, the authors are able to focus on sub problems within data integration within the context of these semantic web services: web service discovery and composition. The former refers to the ability for a data integration system to find the appropriate resource for a given query whereas the latter involves composing an appropriate service from sub services in the form of an acyclic digraph to answer a query. The prime distinguishing factor between the two is that, for discovery, there exists a single service that can answer the query whereas for the latter the system needs to compose a service from other services.

However, in a world of competing standards, such assumptions regarding the structural definitions of an ontology isn't necessarily safe. There are multiple ways that one may define an ontology withing the context of the semantic web. The authors of [33] assume that the semantics are available through the Web Ontology Language (OWL). For the purposes

of a more general approach, the authors of [34] implement an abstraction between the web services and the semantic definition itself. Through introducing the concept of SAWSDL, the authors allow for transformations between classes defined within different ontology domain languages (e.g. an OWL schema vs a RDF Schema).

AT the time, these works were really adopting the model of the semantic web into the context of data integration. However, as we stated before, the industry has moved more toward the model of REST-ful web services that do not necessarily provide the taxonomy and inference rules by which [33] and [34] operate over. In this respect, Lion can be thought of as a first step toward this ecosystem.

Chapter 3: DESIGN AND IMPLEMENTATION

We know turn our attention to the engineering of the titular system: Listen Online, or Lion for short. Lion is a querying mediator for web services that do not provide a machine-ingestible description of their capabilities. Lion is focused on solving the following capabilities within the context of web-services:

- A common data model: Web services may return data that is encoded in a non-unified format, namely XML or JSON. If a system wants to treats these the same, it needs to provide an abstraction that can work in a general way
- A common method for discovery, representation, and interpretation: For the discovery of web service resources, a search on a search engine yielding documentation is presently the method by which they can be found. Not only is this process not easily programmable, it also necessitates that the party performing the search possesses capabilities to effectively interpret the documentation to identify the processes used and data representation.
- A common representation for fetching data and performing operations on this data: Given that REST API's typically describe a method by which to access resources, not necessarily how to structure the requests, a federated system on top of web services needs to concern itself with how to abstract what a request looks like and how to provide functionality on top of these requests that doesn't break what structure the underlying service requires.

As stated previously, a prime consideration regarding the development of Lion is to handle common representations for web services: discovery, data model representation, interpretation, and querying. Lion isn't just focused on providing this, but providing this to a specific class of users who may lack the resources to effectively utilize web services. The canonical example of a user that we worked with during development (and grant proposals) exemplified the following characteristics:

- A researcher who was interested in the data available in a web service. For example, a Professor of Psychology might be interested in data on a social media platform, or a public health research might be interested in health data from services that the public uses.

- They lack programming resources. Whether this is having the ability to write programs or access to individual who do, we importantly emphasize that the users cannot be required to write programs or work with models that are too programmatic in nature.

- They want a structure for consuming data and sharing it. Whether the data is to be used for knowledge generation by way of identifying trends or performing aggregation, or for more complex use cases such as sharing the data with other systems.

With this template of a user, we identified that a primarily graphical interface was an extreme requirement. In addition, we identified that how data was represented needed to be in a structure that focused on entities an their relationships. Lastly, the accessibility of the system from the perspective of a user with respect to how they would access it was a concern.

To address these concerns, Lion is a web application whose availability is limited only based off of the availability of its online host. This is not only familiar to most users, but supports the graphical nature that Lion needed. To address the concern for representation, Lion doesn't directly operate on web services, but instead wrappers in the form of GraphQL servers. Although this decision isn't immediately friendly for a user (which we will address shortly), it addresses several concerns stated above regarding integrating these sources:

- Data model: GraphQL represents all data as JSON. During the resolution process of a GraphQL server the programmatic definition for the types can provide what ever transformation is necessary to convert data from a web service into the schematic representation and then return it as JSON.

- Fetching: Given that the web services are implemented using GraphQL, this provides a standard method by which to perform queries to the underlying web service in the form of a GraphQL query

- Web Service Representation: If a web service is represented using GraphQL, it's representation and interpretation can be solely in terms of the GraphQL specification. This entails not only introspection (for synthesis of the underlying schema), but modeling the data strictly in terms of relationships and attributes, both of which can have contextually relevant descriptions.

There are other concerns such as authentication/authorization, rate-limiting, and discovery of the underlying service, we will be addressing those later. For now, we need to address the concern of user interaction with respect to the GraphQL servers available to Lion. Firstly, we don't expect that a user will necessarily know how to build GraphQL queries as it is a

language that is oriented toward developers first. Secondly, we need to address some limitations of this language and to do so we provide a visual querying language that is more powerful than GraphQL.

3.1 INTERACTIVE GRAPHQL QUERY BUILDING AND CONTROL

As stated previously, we don't necessarily require that our users need to know how to program or work within the context of GraphQL's query language. Instead, Lion provides a novel (to the GraphQL open source community) programming component for building GraphQL queries. This component, composed of two sub components, displays a tree by which users, through interacting with nodes, build a GraphQL query with relatively the same capabilities as the GraphQL language. We say relatively here because, in its current incarnation, it is limited strictly toward queries that don't perform introspection, precluding mutations and subscriptions.

To accomplish this task, we developed three separate web components using the Javascript framework Vue.js [35]. Vue.js is a framework whose intention is to create distinct units of HTML markup with logic and styling (called components) that can be shared composition within the visual GraphQL query builder:

1. A tree-layout component
2. A form-generator component
3. A GraphQL query builder component

3.1.1 A tree-layout component

To understand the need for this component within Lion, we need to understand the model of GraphQL schemata and queries with a little bit more depth. For GraphQL schema, a query always begins at a designated Query type. From that node, relationships extend to other types to designate that one can query for that type. Hereafter, the queries can then extend off of the relationships that are defined in the schema. From a graph theory perspective, this representation is thought of as an acyclic, rooted digraph. At every level within the query, each traversal yields a logical data type that carries semantics with respect to the traversal.

As an example, Figure **??** showcases an example query from Twitter. In one path, a user can query for users of the system and retrieve attributes regarding those users. In the second

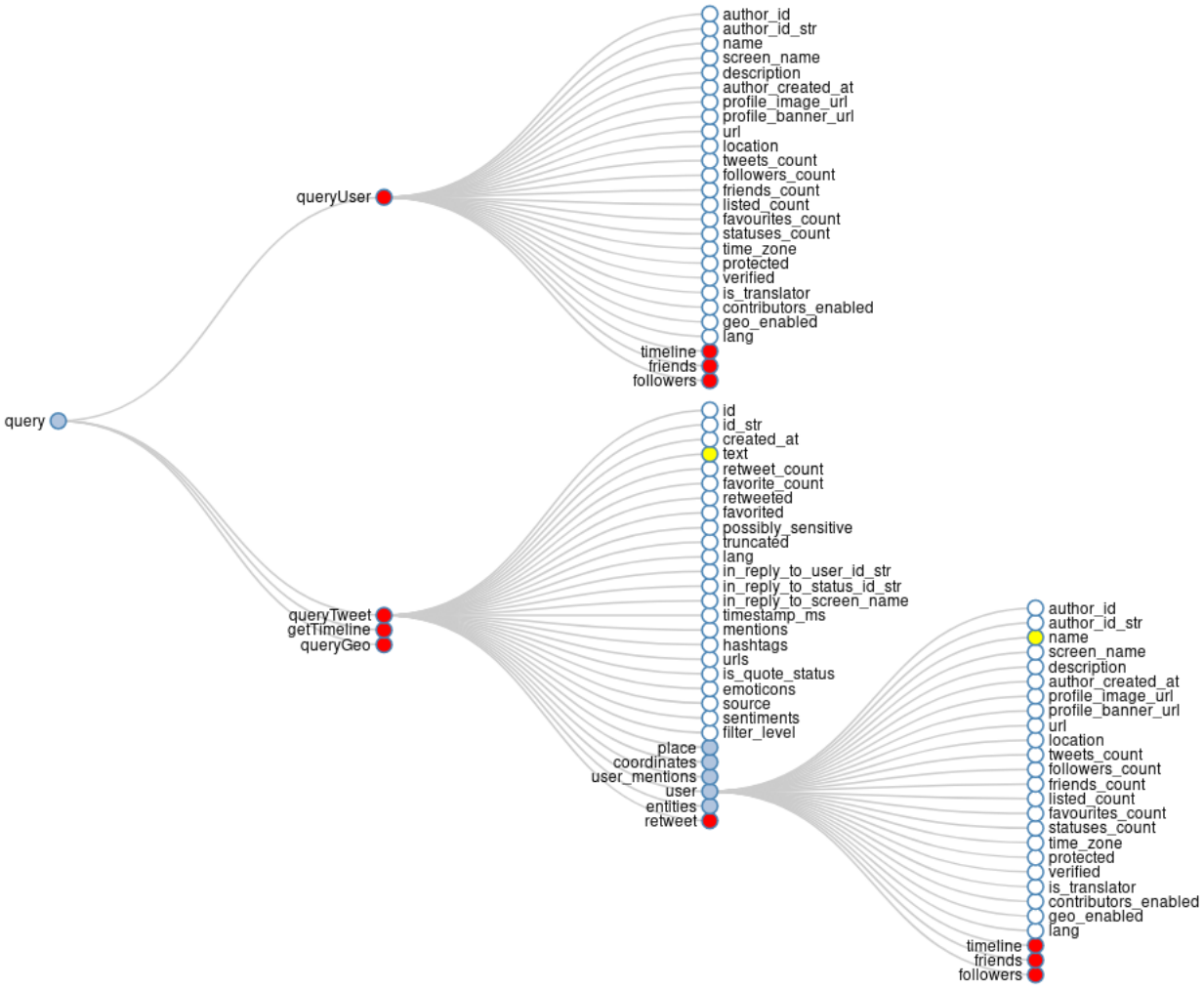

Figure 3.1: An example visual query built using Lion. Here there are two "paths" that show ways to get to a user within Twitter. The first path immediately searches for a user using the "queryUser" attribute of the query. The second path first searches for tweets using the "queryTweet" attribute of the query and then fetches the user who made the tweet.

path, a user can query for a tweet and then continue the query by fetching the associated user who made the tweet. In both cases one can fetch the attributes about a user but how one gets to the user is incredibly important. In the first path, queryUser, what is returned is just a user of Twitter. In the second path, what is returned is the author of the Tweet. This allows one to, hypothetically, fetch tweets matching some search parameter, fetch the author, and perform analysis on all said author's tweets.

The representation of a acyclic di-graph lends itself visually to a well studied problem: tree drawing and layouts. A query can be thought of as a tree, with children representing the relationships traversed within a GraphQL query itself. Thus, in order to visually build GraphQL queries, a problem is motivated that we need to have a way of displaying a tree in a web browser. This tree needs to be visually appealing from a layout perspective (ie the spacing of nodes and presentation of edges) , have capabilities for coloring and labeling of nodes, as well as enable interaction from a user.

To build this, we leveraged the Javascript framework D3.js /cited3.js. D3.js is a framework that focuses around building Data Driven Documents (hence D3) that can display data in a beautiful and interactive way. For D3.js, there is a programmatic way of laying out trees utilizing the Reingold-Tilford "tidy" algorithm [36] that meets our criterion of the visual layout. For the criteria of coloring and labeling, as well as interaction from a user, the need for a component on top of this framework's capability becomes apparent.

The component we developed can be found here [37], a Github repository available online. The key features that enable the other two criteria for this component are accomplished in two ways:

1. Capabilities for coloring and labeling: The component is passed a JSON object describing the tree where each node has properties regarding color, text, and location for the text. Each node is colored the way the calling program describes in JSON and the text is displayed the same way as well, with the location dictating where the text is laid out relative to the node.

2. Enabling interaction from a user: D3.js provides a structure for handling different events that are propagated in a programmatic way, such as clicking and scrolling. To provide a programmatic structure for the tree's interaction, this component captures click events on nodes and emits the event with the node that was clicked. This method allows parent components to integrate business rules regarding clicks without muddling the code-base of the tree-layout component. In addition, this method is idiosyncratic to both Vue.js and D3.js regarding event handling.

Figure 3.2: An example form generated from an argument definition of GraphQL. This definition stems from the "queryUser" node in Figure 3.1, showing the different arguments a User can provide for the search as defined by the GraphQL server.

3.1.2 A form-generator component

The need for a component to generate forms stems from the nature of user interaction during query building for GraphQL. The GraphQL spec describes that relationship traversals may have arguments necessary to be accomplished. These arguments can be on any relationship and are important to the data integration process for web services. For example, say we have a schema on top of a web service for Twitter and a query that we can perform is to search for tweets. This search needs to have a query parameter that Twitter uses to perform the search, such as a hash tag or tweet body content. In the schema, this would be represented as a relationship between the query type and tweet type having a "query string" argument.

In practice, web services typically allow for many differing parameters that control the response from the service. These can be a query string, an integer for a limit of response objects, parameters regarding the date these objects were created, etc. Thus, in order to allow for these arguments to have values for the query building component, we needed a component whose input would be an argument from a GraphQL schema, and would create an appropriate form whose input elements reflected the data types of the arguments themselves. These data types can be Scalars (Strings, Integers, Floats, Booleans) or Object types whose attributes are composed of Scalars.

Thus, for the purpose of user interaction, a form-generator component was developed and can be found here [38], a Github repository online repository online. This component is fairly simple, mapping scalars to their appropriate HTML input types (Strings to text inputs,

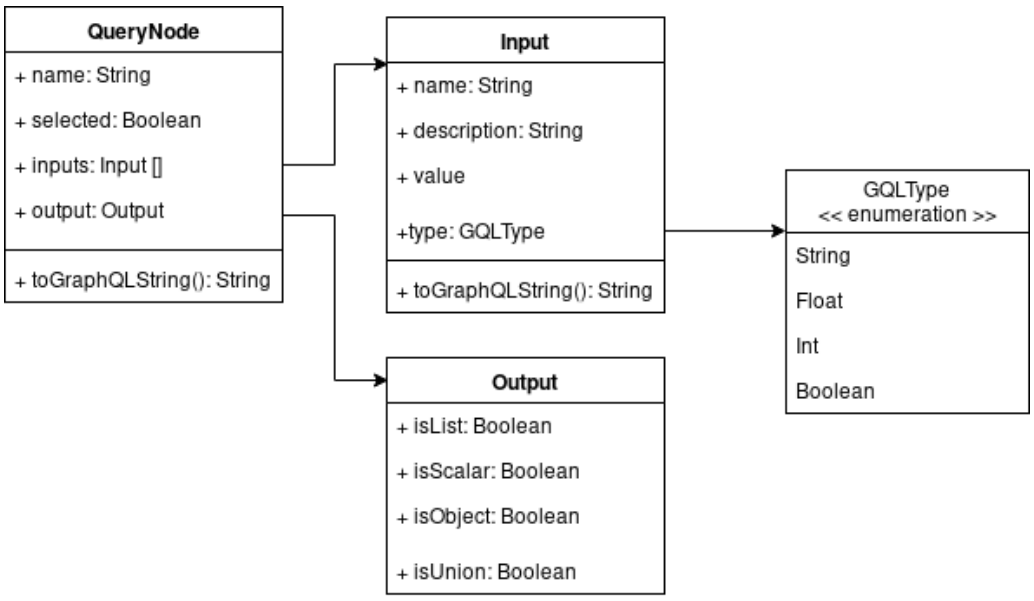

Figure 3.3: A UML diagram of representing a GraphQL query as a data structure in memory.

Integers and Floats to number inputs, and Booleans to check box inputs). This component makes use of the argument definition in order to display contextually relevant information (ie description) and load default values for these inputs. In terms of interoperability with other components, this form-generator makes use of one-way data binding to synchronize the changes to the input values to the data representation that is passed to the component. Lastly, for the purpose of Object type inputs, this form is recursive in that it creates sub-instances of itself to capture object types.

3.1.3 A GraphQL query builder component

The GraphQL community already has a popular resource for interactively building text queries in the form of GraphiQL. This software utilizes GraphQL's introspection functionality to provide autocomplete capabilities for writing queries. However, as stated previously, our focus is on enabling visual query building. Thus, this software was insufficient for our needs and there was no comparable library to visually build queries. As such, we built a visual GraphQL query builder utilizing the aforementioned tree-layout and form-generator components that we built, as shown in Figure 3.1 and available online here, a Github repository.

These components enable a visual representation of a query, but that visual representation ultimately needs to be able to be mapped to it's corresponding GraphQL string representation. To do this, we developed an object oriented model that enables us to map the query

that the user creates visually to a GraphQL string shown in Figure 3.3. Here we refer to each type in a GraphQL query as a `QueryNode`. Each node has a `children` attribute on the `QueryNode` refers to the attributes that are available given a GraphQL type. These represent attributes that can be scalar types, object types, or combinations thereof. The names of the node refers to the name of the type's relationship relative to its parent. As an example, say a `Tweet` object type has an `author` property referring to a `User` object type. Here, the `User` corresponding `QueryNode` relative to the `Tweet` would have a name of `author` that is labeled appropriately on the tree.

The selected attribute on the `QueryNode` refers to whether a user of our system intends for that type to be included in the results of a query. This is useful for displaying of the type in that types can have many relationships with other types but we only desire to actually show the relationships if they're part of a query. In Figure 3.1, we describe the nodes "queryUsers" and "queryTweets" as being selected.

The inputs array on a `QueryNode` refer to the arguments necessary to traverse that relationship. The form-builder component mentioned in section 3.1.2 consumes this property in order to build the form itself. Note that for the `Input` object definition we make use of our implementation in Javascript/PHP to have attributes that are dynamically typed in the case of the `value` and `type` attributes. This attribute is useful because we can encode information in the displaying of a node based on whether it requires arguments or not. In Figure 3.1 "queryUser" and "queryTweet" both require arguments and, as a subjective design decision, display it as red.

Lastly, the `QueryNode` has an attribute describing the "output" of the node. What this represents is that nodes themselves, when queried, have a varied output as per the GraphQL spec. A node can be a object, union, scalar, or lists thereof. In order to provide contextually relevant information during the visual query building process this attribute is leveraged for choice of node color and labeling. In Figure 3.1 we choose to display nodes who are object types as either light blue or red (depending on arguments), with the text on the left side of the node. Nodes with scalar outputs can no longer be expanded and, as such, we display the text on the right side of the tree and show the node as white if it is unselected, or yellow if it is selected. In FIgure 3.1 the attribute "text" of a tweet and "name" of the author of the tweet are selected scalar outputs.

The effects above regarding displaying the tree doesn't necessitate the object composition described in Figure 3.3. All of that information is not only available through introspection of a GraphQL server which is precisely how the query builder functions. Instead, the primary utilization of this structure is for mapping the query to a GraphQL string. This entails both a check to ensure that what the user has selected can be translated into a valid query string

and the process thereof. The component that we developed to build GraphQL queries can be found here [39], a Github repository online.

Algorithm 3.1 An algorithm to test if a built query is valid, VALID

```
Input: A QueryNode root
Output: A boolean representing if that query is valid.
if ! root.selected then
    return false
else if root.output.isScalar then
    return true
end if
valid ← false
for all root.children as child do
    valid ← valid || VALID(child)
    if valid then
        return valid
    end if
end for
return valid
```

Algorithm 3.1 is the algorithm that we use in order to test whether or not a visual query is valid in its current state. This algorithm is a variation of a depth first search on the tree that searches the leaves in each subtree to make sure that they are scalar and are selected. Note that the algorithm itself will view partially correct GraphQL queries as valid. Here, the definition of valid is such that there exists at least one subtree that contains a selected scalar attribute. Based off of this, we can convert some partially valid query into a GraphQL query string utilizing Algorithm 3.2. An example in Figure 3.1 would be the queryTweet tree, which can be transformed into a GraphQL query, whereas the queryUser tree would be invalid.

Algorithm 3.2 too is a variation of the depth first search of a tree. At each sub tree we convert the in memory representation in Figure 3.3into the grammar of GraphQL. From a design standpoint, making each `QueryNode`'s name relative to its parent (as oppose to the name of the type, such as `User`) means that when we can simply use the name of each node. In addition, during the query construction process the parents of each node are the ones that actually create its children. This enables the name to be flexible for union types whose syntax within the query is slightly different than the base case of the attribute name of the parent.

Within Lion we give the user the ability to run queries themselves, but we felt as though that was insufficient for identifying trends over a long period of time. It would be incredibly

Algorithm 3.2 ToGraphQLString

```
Input: A QueryNode root
Output: A string representing a valid GraphQL query.
if ! root.selected then
    return ""
end if
queryString ← root.name
if root.inputs.length > 0 then
    queryString ← queryString + "("
    for all root.inputsasinput do
        queryString ← queryString + input.name + ": " + input.value + " "
    end for
    queryString ← queryString + ")"
end if
if root.children.length > 0 then
    queryString ← queryString + "{"
    for all root.childrenaschild do
        queryString ← child.toGraphQLString()
    end for
    queryString ← queryString + "}"
end if
return queryString
```

inconvenient for a user to have to log in based off of what ever time granularity they would like to perform the queries. As such, after a user creates the query, we persist both it's structure as a`QueryNode`, as well as a user provided name describing the query and a schedule as to when they want the query to run.

3.2 META QUERIES

Now that we have the capabilities for performing GraphQL queries we need to discuss why only providing facilities for GraphQL queries is insufficient for our users. Firstly, most sophisticated query languages provide functionality for performing simple amounts of aggregation. The specification for GraphQL does not support this capability, making simple concepts such as counting the responsibility of either an underlying attribute on a schema or a consuming program. Given that the consuming program of Lion only seeks to query the data, we've got an issue that we need to resolve to provide aggregation.

In addition to aggregation, during the development of Lion we found that the capability of performing a "join" relative to a series of queries a necessity. This notion of a join

isn't in the relational sense but more of an input-output sense. We want the capability of performing a query and utilizing an output set of data as inputs to another query. The go-to example that we used in development of Lion was finding hashtags on Twitter and, subsequently, performing a search on Youtube for videos. For the purposes of Lion, we treat each GraphQL server as a separate entity whose schema we may or may not be able to change. If we want to achieve the above example, Lion needs to have a structure that supports joins.

Lastly, as a query language, GraphQL doesn't provide capabilities that are friendly for selection. As an example, say one were only interested in which hashtags were used in a given query for tweets. Unless the underlying GraphQL server supports such an attribute on a Tweet (a mutation of the schema that, again, we don't assume we can provide) then the GraphQL grammar and, by extension Lion, can't be used by itself without returning duplicate values . We need to provide a method for performing filtering and selection which GraphQL simply lacks.

With the need for a higher level of abstraction for queries we now begin the foundation for what we refer to as a "Meta Query". Meta Queries, on a high level, are an orchestration of GraphQL queries, previously saved query results, functions on top of those results, and, recursively, other Meta Queries. Meta Queries function by treating each of the above categories as a logical Node within a acyclic di-graph. These nodes are represented effectively as a black box of strictly inputs and outputs, which we'll delve into respectively:

3.2.1 GraphQL queries as a Meta Query Node

We already showed an in-memory data structure representing a GraphQL query in Figure 3.3. In order to represent this structure as a Meta Query Node, we need to have an unambiguous representation for inputs to the query and outputs to the query. For example, in Figure 1.2 we show an example query where on relationship has an argument of *id* and the attributes that we select on the droid and hero respectively are *name*. In this case, treating the inputs as just "id" and outputs as "name, name" is confusing from the perspective of a user. As such, for GraphQL queries, we choose to define the inputs and outputs as paths through the query structure. Here we use the term path as a finite sequence of edges that connect the query type to the distinct verticies of the output scalar values.

For our in memory representation of a GraphQL query, Algorithm 3.3 and Algorithm 3.4 enumerate over all the inputs and outputs within a given GraphQL query to give us an unambiguous representation. For example, the query in 3.1 is represented by the followings input paths :`query.queryTweet.q, query.queryTweet.count, query.queryTweet.pages,`

Algorithm 3.3 An Algorithm to generate all of the input paths of a given query, getInputPaths

```
Input: A QueryNode root
Output: A list of strings representing input paths of a query.
if ! root.selected then
    return []
end if
paths ← []
for all root.inputs as input do
    paths.append(root.name + "." + input.name)
end for
for all root.children as child do
    childPaths ← getPaths(child)
    for all childPaths as path do
        paths.append(root.name +"." + path)
    end for
end for
return paths
```

`query.queryTweet.geocode, query.queryTweet.result_type, query.queryTweet.locale, query.queryTweet.until, query.queryTweet.lang`, all of which are arguments that are needed to retrieve tweets using the `queryTweet` relationship.

For the output paths of Figure 3.1, we have the following: `query.queryTweet*.text` and `query.queryTweet*.user.name`. Here we can understand quickly that we'll have the same number of user names as the texts of the queries by referencing the '*' character appended to the `queryTweet` relationship. If we were to also grab the user's timeline through an example output path as `query.queryTweet*.user.timeline*` we know that for each user we would have many timeline tweets.

3.2.2 Query results as a Meta Query Node

Representing a queries result as a meta query node is far simpler of a process. These results are just responses from the underlying web services and are stored in some database. As such, they strictly have no inputs and its outputs are the paths from the query that generated the result. In terms of its usefulness within Lion, the ability to have a Meta Query Node be previously fetched data is two fold:

1. We don't assume that the underlying web service will consistently return the same results for requests. The nature of some web services (particularly those that are

Algorithm 3.4 An Algorithm to generate all of the output paths of a given query, getOutputPaths

Input: A QueryNode *root*
Output:A list of strings representing output paths of a query.

```
if ! root.selected then
    return []
end if
nodeEdge ← root.name
if root.output.isList then
    nodeEdge ← nodeEdge + "*"
end if
paths ← [nodeEdge]
childPaths ← []
for all root.children as child do
    childPaths.concat(getOutputPaths(child))
end for
for all childPaths as path do
    paths.append(nodeEdge + "." + path)
end for
return paths
```

around users and generated content) is volatile and, as such, a query could return different values, even given the same parameters. Thus, if a user in particular likes the data made from a previous query then they can choose to use that data instead of having to perform a new request.

2. If part of a Meta Query fails,then by allowing for a previously generated result to be used then it enables us to restore a long running query from a failure. An example of this could be when performing many requests to a single web service, we hit a rate limit issue preventing subsequent requests from completing successfully for some predetermined about of time. In the face of this we can still save previously generated results and continue the execution of a Meta Query after waiting for some amount of time.

As an example of this, the result of a query could be JSON as shown in Figure 1.2. Here the inputs to the node would be an empty set as we would not be changing the data after having queried it. The outputs to that node would be: `data.hero.name` and `data.droid.name`. As a note here, we better distinguish between utilizing a Query Result as a Meta Query Node from a GraphQL query by presenting the output path as rooted from a data node as opposed to a query node.

3.2.3 Functions as a Meta Query Node

The useage of functions as a Meta Query node is where the visual query language of Lion becomes more powerful than that of GraphQL. We define that any function can be made as part of a Meta Query. We classify functions within Lion to be part of three categories:

- Transformative: Transformative functions are those that do not necessarily increase the dimensionality of an input set of values, but apply some transformation on them. Examples of these functions within Lion include: capitalization of strings, sorting, a limiting of values (such as limit 5, 10, 25, etc), filtering, and basic arithmetic operations applied to each numeric value.

- Dimensional Reductive: functions that reduce dimensionality are typically those that aggregate. Examples of these functions within Lion include: max, min, average of numeric values, and counting of a set of values.

- Dimensional Expansion: functions that expand the dimensionality of a set of values. These can be useful if a user wants to perform analysis at a more granular level. An example function that expands dimensionality within Lion is to split a string based off of white space.

Within the architectural definition for functions within Lion, any program that consists of strictly inputs and outputs on a set of values can be used as a Meta Query Function. This is incredibly flexible and makes the process very friendly for administrators of Lion to add functionality. For the inputs and outputs of these functions as Meta Query Nodes, the inputs are typically typed to avoid logically ambiguous operations (such as trying to split the white space on a set of integer values) and their outputs also describe the effect that the function has on the dimensionality. For example, a `Sum` Function as a Meta Query Node might have an input of `valuesToAdd*` that accepts numeric values and an output of `sum` as a numeric value.

3.2.4 Meta Queries as a Meta Query Node

Meta Queries treat their own nodes as a set of inputs and outputs. One can also encapsulate a Meta Query as a set of inputs and outputs as well. The inputs of a Meta Query are simply the inputs of all nodes within the Meta Query, with the outputs being the same. This recursive structure is useful for a user in that they can reuse the Meta Queries that

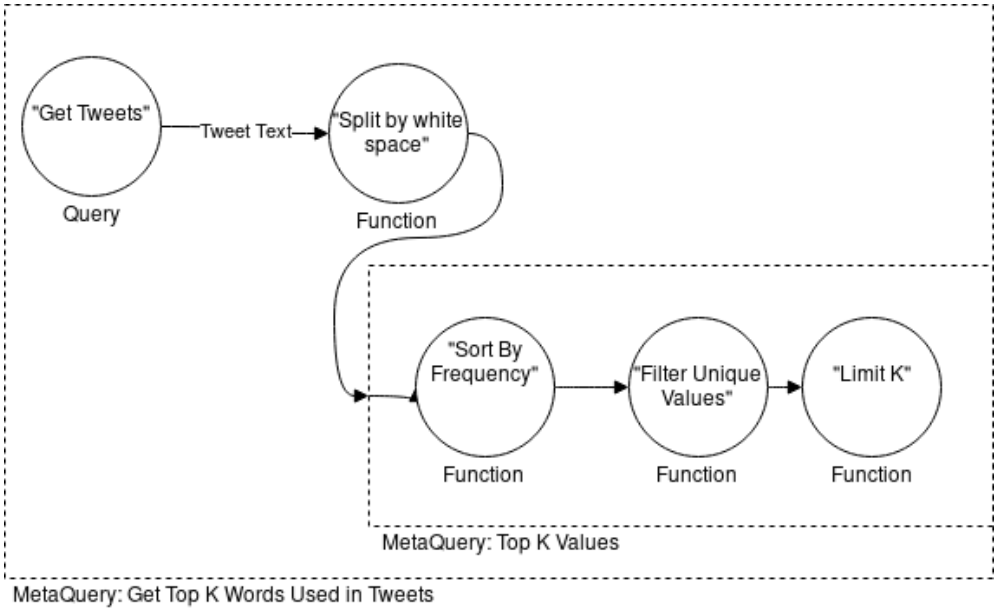

Figure 3.4: An example composition of Meta Query Nodes using a Query, Function, and Meta Query.

they have previously saved, but compose interesting functions even if their instance of Lion does not support them.

As an example, a common type of query to perform on a set of values might be to find the top-k most common elements. A developer could easily add a top-k function to Lion given the architecture for functions described previously, but a user could also compose such a function out of many smaller functions as well. A set of functions that would mimic a top-k function, that we have very commonly used in demonstrations of Lion, would be: sort by frequency -¿ filter for unique values -¿ limit k. Thus a user can create Meta Queries as a composition of functions and then reuse their work easily. An example of this can be found in Figure 3.6 where a "Get Top K" Meta Query is used in tandem with a composition of queries and functions to retrieve the set of words used in a search for tweets. The representation for inputs and outputs of a Meta Query is a recursive operation that simply retrieves the inputs and outputs of its child nodes.

Now that we've established the different kinds of logical elements that we've used for Meta Query Nodes, we can now begin to define the behavior of a Meta Query itself. Meta Queries are composed of Meta Query nodes and dependencies between these nodes. By doing so, we can support our desired functionality of selection, aggregation, and joins. Figure 3.5 showcases the UML class composition that describes Meta Queries.

During the process of Meta Query creation, a set of nodes participant in the query are

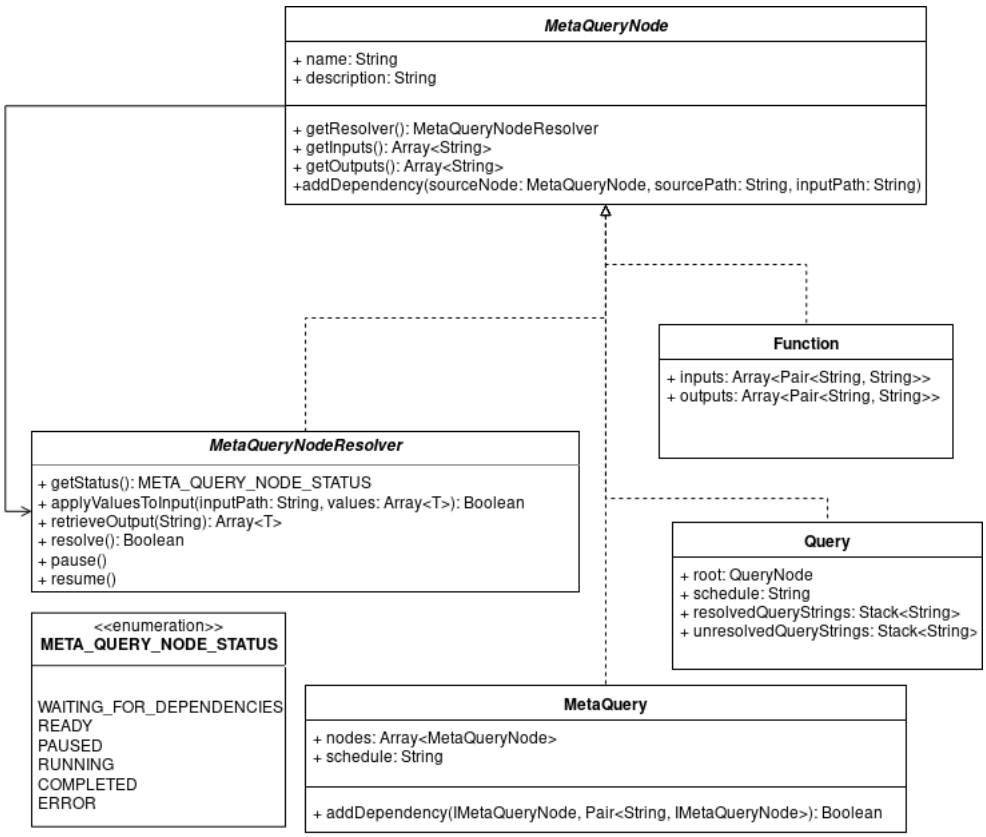

Figure 3.5: A UML class diagram representing the components of a Meta Query.

added by the user in a visual way shown in Figure **??**. The user adds edges from the output of nodes to the inputs of other nodes, creating the dependency graph visually and registering the dependencies with the Meta Query Node. Note that in Figure 3.6 we concatenate the output types to the paths so that it's easier to distinguish which outputs and inputs share compatible types. In addition to the impeccable choice of colors for these Meta Query node boxes (as specified by the Illinois Identity Standards), these nodes are interactive in that a user can freely move them about a canvas, create lines solely by clicking, and work strictly in terms of these boxes.

In terms of the order of resolution, a topological sort encoded within the composition of classes resolves nodes that have all their dependencies available. To resolve each kind of node, it must provide some kind of class that is capable of resolving that node in the context of the Meta Query. This is an important decision within Lion because the behavior of each kind of Meta Query Node with respect to resolution is highly dependent on the type. For example, to resolve a function is a straight forward process where you take a list of values and apply the function to them. For the process of resolving a Query or Meta Query, there is much more work involved.

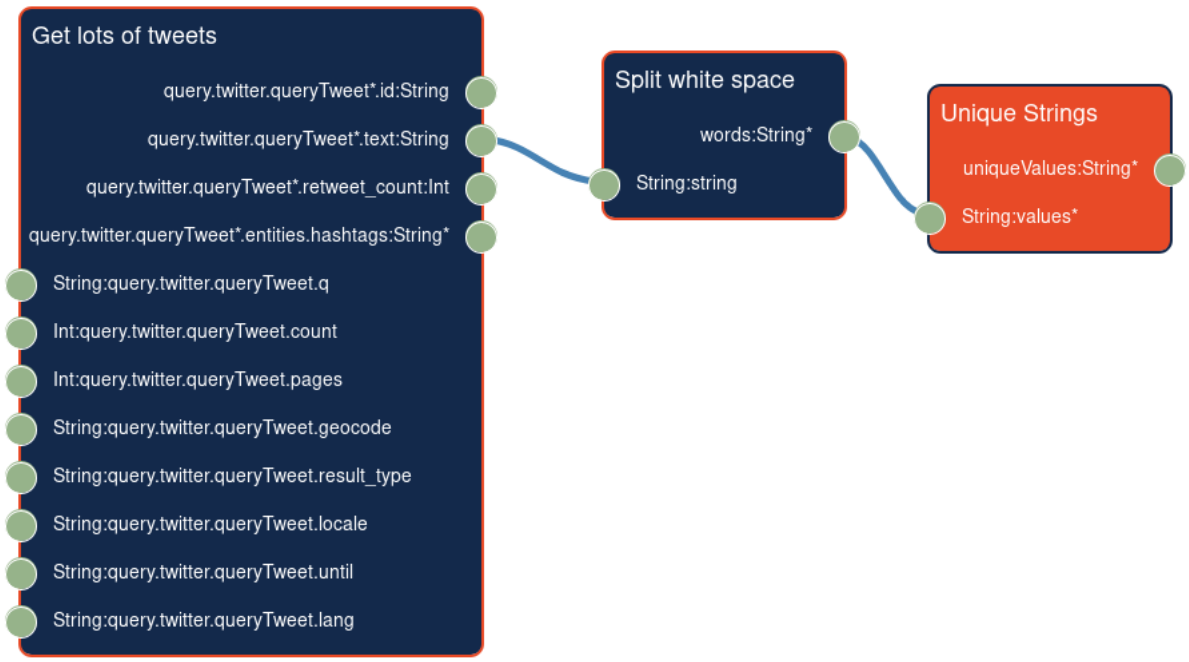

Figure 3.6: An example Meta Query construction using the current UI of Lion.

For a Query, the act of joining a set of values to the inputs to the query is a multiplicative effect where for every value to an input, there are a corresponding number of query strings that result. As an example, say a Query had inputs that were being joined on with 2, 5, and 10 different values. Ultimately, this results in a total of $2 \times 5 \times 10 = 100$ query strings that need to be resolved. If the resolution process is interrupted or paused, it's up to the resolver in question to ensure that it keeps track of which queries have been resolved and which ones have not. The way that it handles this is by firstly storing these query strings to database and by keeping trick of which ones have and have not been resolved.

A Query operates in a binary form of success: either the query is able to be completed or it fails for myriad of reasons. Common reasons why a Query fails are either the user input doesn't respect an input format requirement (Such as date formatted as dd/mm/yyyy versus a requirement of mm/dd/yy), the GraphQL server that the query string was destined for being down, or the query hitting a rate limit. If a single query string fails, our solutions are fairly limited. The first two are technical issues that can be resolved by addition of mediators that can match formatting between queries, but the last issue is one that wholly depends on implementation of the underlying server (which, again, Lion makes no assumption that the authors can change the implementation).

Meta Queries are entirely a different story however. With a Meta Query, the intermittent nodes themselves may vary in their state from those described in the `META_QUERY_NODE_STATUS`

enumeration of Figure 3.5. When a Meta Query is processing it's intermittent nodes, a Query node may be able to fetch data for some of the queries that it performs, but ultimately hit some form of error during processing. A Meta Query's job, at this point, is to effectively pause that node for a period of time that would enable it to finish its job. These nodes can receive a multitude of inputs, spawning hundreds if not thousands of resulting queries/web requests that all need to be resolved to continue.

Multiple strategies exist for this kind of problem. One that comes most immediate to mind is to view this as a resource scheduling and utilization problem. Another solution could be to train a model on the behavior of these queries with respect to their GraphQL server and see how they behave for Meta Queries that create many sub-queries. Ultimately the current implementation of Lion uses a heuristic of "wait 5 minutes and try again" which, although not necessarily the most sophisticated, presents an immediate solution and leaves an avenue for further research.

3.3 DATA INGESTION

Lastly, the engineering Lion does not assume that a user will only interact within Lion for performing analytics. Although Lion provides a general structure for functions, which could perform interesting analytics, Lion also provides two methods by which to share data with other applications. The first method by which Lion exposes data is through the use of a "callback" system. Lion enables users to register "callbacks", or a URL that Lion will make a post request to whenever they perform a Query or Meta Query. The user merely specifies a URL, name for the consuming application, and a description by which they can refer to the application. This is a nice convenience as the consuming application doesn't necessarily need to concern itself with how Lion persists that data within it's own database.

The second method by which Lion operates is, delightfully, is through a GraphQL server. Here Lion exposes information that it has available regarding querying: the GraphQL servers available, the users, their queries/meta queries, the functions available, etc. This design enables a consuming application to consume the exact data that it is interested in, as opposed to all data that a particular user generates. A consequence of this is that Lion is, by extension, a service that is available to query through Lion.

In addition to the above methods for sharing data, Lion also promotes the ability to perform queries without user intervention. This is, perhaps, less interesting of an engineering challenge beyond the handling of authentication and authorization tokens. How Lion approaches performing querying without user intervention is by allowing a user to define a schedule for Queries and Meta Queries. With an underlying implementation of cron expres-

sion, there is flexibility for almost any granularity for time between a minute and a year. Given our class of a user, however, we currently choose to allow them to set a query to run once a minute, once an hour, once a day, once a week, or once a month. Lastly, we allow them to not have a schedule and only run the query when the user themselves manually runs the query.

For the challenge of handling authentication and authorization, Lion acts as a monolith. It is registered with different web services (as applicable) and persists authorization tokens. When a request is made to a designated server, a mediator in software is passed the token and structures the subsequent request to the GraphQL server in the method by which it is expecting. Although the authors agree here a designated mediator design would be far more superior to reduce coupling, the current methods by which authorization/authentication is handled for most applications is insufficient for Lion's architecture.

Chapter 4: DISCUSSION

4.1 WHAT IS LION?

Within the context of data integration, we can classify Lion within the architecture of data mediators. Lion acts as an orchestration of mediators by providing indexing of the available GraphQL servers, a language to orchestrate queries that span multiple web services, and mediators that handle the question of authentication and authorization for accessing the web services.

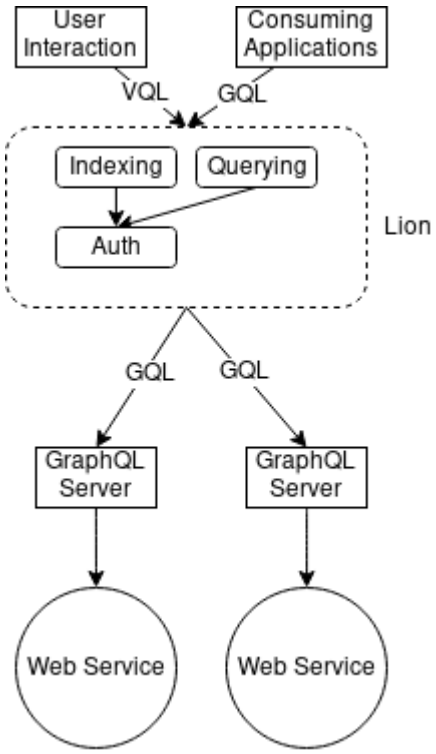

Figure 4.1: A diagram showing the mediation structure that Lion provides.

Figure 4.1 showcases the mediation interaction that Lion provides for interaction of web services. Lion provides mediation for indexing of available web services, querying of these web services, and handling of authentication/authorization for these services. For the interaction of Lion as a mediator, we concentrate on the following capabilities:

- Mediating access to Lion with users and client applications
- Providing indexing of GraphQL servers,

- Providing mediation of queries made in Lion's visual query language (VQL) for GraphQL queries and Meta Queries into their corresponding GraphQL queries,
- Providing mediation between a query made to a GraphQL server and the authorization information that it needs.
- The utilization of GraphQL as a wrapper and mediator for a corresponding web service.

Mediating access to Lion with users and client applications means that Lion needs to provide an interface to the capabilities it provides following a standard language. For user's, they have the capabilities to create queries, meta queries, add callbacks, run said queries, and view the data. For client applications, Lion exposes the queries that are available, the data generated from said queries, as well as an index over the GraphQL servers that it knows are available.

For providing said index, Lion makes use of GraphQL's introspection capability that lets a server describe itself via GraphQL. The query that Lion uses to do this can be found in Appendix A.1. This query describes the different kinds of queries that the server supports (e.g. the query type, mutation type, and subscription type), the directives available, as well as all types and their relationships.

Lion handles the mediation of queries built utilizing the visual query language by making uses of the visual query builder and meta query builder as described prevoiusly. These are represented and stored in a database within Lion as their object oriented representation and then materialized as the appropriate GraphQL query strings. For Meta Queries this process involves mediation between querying with querying itself through the application of output values to input parameters of queries.

For authentication and authorization Lion provides a mediator between services and the means by which they validate user permissions. These GraphQL servers provide a contract by which this process operates (e.g. a token in a specific header) which Lion provides a mediator, as applicable, for. As an example, an arbitrary GraphQL server may require some authorization token for permission to query. This server, when sent a request without any tokens for permission, would return some http error. To avoid this Lion encapsulates mediators for requests made to that specific server to retrieve tokens from its database and structure the request in the exact format that the server requires. For acquiring the tokens themselves, Lion needs to be registered with the underlying web service as a web application that consumes it. Although this partially breaks the abstraction that Lion operates on, there's presently no great way following the Oauth2.0 spec that enables a similar functionality to be isolated to a wrapper.

With GraphQL operating over a web service, it provide the functionality of a wrapper and a mediator. Firstly, the GraphQL schema over a web service provides a wrapper around the logical entities such that they can represented within the conceptual schema of a user's operation. Whether a logical entity is available through a single web request or a series of web requests the GraphQL schema and underlying representation effectively insulates a consuming service from how that entity is constructed. This presents a lot of flexibility on the definer's end as they can present whatever level of granularity is necessary or available.

Secondly, by defining the wrapper around the web service as a GraphQL server, we also define a mediator for that wrapper's query language through the GraphQL specification. That is, we are no longer concerned necessarily with the underlying language that the web service operates on but, instead, the language of the GraphQL server which is definitively the grammar of a GraphQL query. Thus, to be succinct, operating on a GraphQL server allows for 1) A wrapper around the API and 2) A mediator for that wrapper that maps the conceptual schemata to the GraphQL query language.

4.2 LION IN THE CONTEXT OF DATA INTEGRATION SYSTEMS

Now we need to put Lion within the context of the prior art. Like the TSIMMIS project Lion operates over wrappers of a source. TSIMMIS' wrapper is in the form of a translator that can handle OEM queries and translate it into the corresponding OEM queries, whereas Lion doesn't translate GraphQL queries into a database query. Instead, Lion translates it into an orchestration of web requests. In terms of who is using the query language, TSIMMIS aims at a certain class of users that can use its SQL-like language. In contrast, Lion's target users dictates a necessity for a interactive visual manner.

In addition, TSIMMIS provides a comprehensive manner by which to impose constraints on the data sources that it integrates. This constraint manager integrates with local constraint constraint manager to ensure that operations that change data follow the constraints as defined within the underlying data sources. By contrast, Lion doesn't approach this consideration as the constraints imposed by the GraphQL server representing a source are, ideally, represented faithfully. In this respect, the GraphQL server acts as a translator and local constraint manager in one.

For the Garlic project, Lion acts in a very similar manner. Garlic represents a layer within an integration system that focuses mostly on querying. The various services connect to Garlic in a fairly loosely coupled way and Garlic's meta data repository provides a listing or index of the sources available. Lion performs a similar job w.r.t. GraphQl servers. Both Lion and Garlic interact through wrappers that need to be manually created. The nature by

which wrappers can be created, as shown in [27] describes a nice object composition method that streamlines the process. In terms of language, Lion too is distinguished from Garlic by its visual nature. Garlic's language, like the TSIMMIS project, is also aimed at a class of user that is far more knowledgeable of systems than Lion's.

For BigDawg, the authors haven't considered the model of web services. Instead, they liken their system as a point of contact by which an application can have an integrated view of the underlying data sources. That is, a calling application can work through the schema defined by the author's island model. This is nice from a caller perspective, but it still limiting in that one requires a connection to the underlying data source. By contrast, Lion doesn't provide as much in terms of capabilities but it does relax the assumption that we have first-class access to the data.

4.3 LION IN THE CONTEXT OF DATA INTEGRATION OVER WEB SERVICES

From the perspective of data integration systems over web services, Lion's approach is more flexible but requires far more work. The authors of [33, 34] both are working on a far more rigorous definition for a web service. Namely, the web services they operate on are those that have a WSDL definition and can be described by an ontological definition. By contrast, Lion assumes that the GraphQL servers it operates on represent the underlying semantics of the web service faithfully. What this means is that, although Lion has a larger build up time to integrate a new source, it is more flexible in the kinds of services it can integrate. Lion only requires that the definition of the schema provides a method to resolve the data. Due to this it can effectively be built on top of a web service that may not necessarily have a WSDL description or ontology. In modern times what this means is that Lion can provide the querying capabilities on top of services whereas the prior art are currently made obsolete due to how the industry has moved.

Chapter 5: FUTURE WORK

Lion is an important first step into this brave new world of data integration for web services that aren't built on top of the ecosystem of SOA, WSDL, UDDI, OWL, RDF's, etc etc. Lion still presently only supports querying functionality that relies on faithful depiction of the there is still a large amount of avenues for future work. We will separate these avenues of work into the following categories: Architectural improvements, functional improvements, Meta Query optimization/cost-estimation/scheduling, and semi-automatic schema synthesis.

5.1 ARCHITECTURAL IMPROVEMENTS

Presently, Lion's architecture bundles many capabilities within a monolithic "mediator" model to support the following: indexing of GraphQL servers, orchestration of Meta Queries, and transformative mediation of queries for authorization. Although this design was useful for quick prototyping, it is insufficient for providing a more general ecosystem by which people can make contributions and new research ideas can be implemented. As such, the author believes a more distributed architecture, even if Lion is treated monolithically, is best to have looser coupling and quicker prototyping for future improvement.

5.2 FUNCTIONAL IMPROVEMENTS

Presently, Lion stands as a query mediator but little else in terms of a data integration system. More mature systems try to provide functionality such as mapping of abstractions, automatic conversion of data types/units, and handling of missing attributes. If Lion can be made to utilize a more distributed architecture as described in Section 5.1 then these kind of functional improvements can be better orchestrated or automatically applied.

5.3 META QUERY OPTIMIZATION, COST ESTIMATION, AND SCHEDULING

Within the querying model of Lion we don't have the traditional costs associated with queries. Instead of the amount of time to retrieve data from disk, or the size of the data retrieved, our model is based primarily around time to retrieve the data from the network and a cost of queries made within the context of a rate limit. As such we have an opportunity within Meta Queries to optimize the query to take as few requests as possible if applicable, as well as to schedule the sub queries to run within the context of these costs. Within that

scheduling includes a more sophisticated strategy for handling rate-limiting errors than the heuristic mentioned earlier.

5.4 SEMI-AUTOMATIC SCHEMA SYNTHESIS

What goes around comes around with this potential improvement to Lion. In previous works for data integration on top of web services the prior art was able to make use of the Semantic web to have an effective schema for free. In the context of the current best practices for web services we don't have such a representation easily available. Instead, we've got a taxonomy that is embedded within the semi-structured text of the service's documentation. Given the nature of this structure, the author hypothesizes that a taxonomy can be generated from said documentation, providing a skeletal structure of a GraphQL server by which individuals can provide resolvers.

BIBLIOGRAPHY

[1] M. Lenzerini, "Data integration: A theoretical perspective," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2002, pp. 233–246.

[2] G. Wiederhold, "Mediators in the architecture of future information systems," *Computer*, vol. 25, no. 3, pp. 38–49, 1992.

[3] G. Wiederhold, "Knowledge and database management," *IEEE Software*, vol. 1, no. 1, p. 63, 1984.

[4] A. Y. Levy, "Logic-based techniques in data integration," in *Logic-based artificial intelligence*. Springer, 2000, pp. 575–595.

[5] M. Gagnon, "Ontology-based integration of data sources," in *Information Fusion, 2007 10th International Conference On*. IEEE, 2007, pp. 1–8.

[6] A. Y. Levy, "Combining artificial intelligence and databases for data integration," in *Artificial Intelligence Today*. Springer, 1999, pp. 249–268.

[7] A. Halevy, A. Rajaraman, and J. Ordille, "Data integration: the teenage years," in *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 9–16.

[8] W. Kent, "A simple guide to five normal forms in relational database theory," *Communications of the ACM*, vol. 26, no. 2, pp. 120–125, 1983.

[9] M. Stonebraker and J. Hellerstein, "What goes around comes around," *Readings in Database Systems*, vol. 4, pp. 1724–1735, 2005.

[10] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000, vol. 7.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," Tech. Rep., 1999.

[12] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple object access protocol (soap) 1.1," 2000.

[13] D. Hardt, "The oauth 2.0 authorization framework," 2012.

[14] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana et al., "Web services description language (wsdl) 1.1," 2001.

[15] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific american*, vol. 284, no. 5, pp. 34–43, 2001.

[16] G. Klyne and J. J. Carroll, "Resource description framework (rdf): Concepts and abstract syntax," 2006.

[17] S. Bechhofer, "Owl: Web ontology language," in *Encyclopedia of database systems*. Springer, 2009, pp. 2008–2009.

[18] N. Shadbolt, T. Berners-Lee, and W. Hall, "The semantic web revisited," *IEEE intelligent systems*, vol. 21, no. 3, pp. 96–101, 2006.

[19] S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[20] "Graphql: A query language for apis." 2015. [Online]. Available: https://graphql.org/

[21] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom, "The tsimmis project: Integration of heterogenous information sources," 1994.

[22] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic et al., "Towards heterogeneous multimedia information systems: The garlic approach," in *Research Issues in Data Engineering, 1995: Distributed Object Management, Proceedings. RIDE-DOM'95. Fifth International Workshop on*. IEEE, 1995, pp. 124–131.

[23] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, "The bigdawg polystore system," *ACM Sigmod Record*, vol. 44, no. 2, pp. 11–16, 2015.

[24] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas, "The rufus system: Information organization for semi-structured data," in *VLDB*, vol. 93, 1993, pp. 97–107.

[25] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom, "The tsimmis approach to mediation: Data models and languages," *Journal of intelligent information systems*, vol. 8, no. 2, pp. 117–132, 1997.

[26] R. G. G. Cattell, D. K. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade, *The object database standard: ODMG 2.0*. Morgan Kaufmann Publishers Los Altos, CA, 1997, vol. 131.

[27] M. T. Roth and P. M. Schwarz, "Don't scrap it, wrap it! a wrapper architecture for legacy data sources." in *VLDB*, vol. 97, 1997, pp. 25–29.

[28] K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. " O'Reilly Media, Inc.", 2013.

[29] P. G. Brown, "Overview of scidb: large scale array storage, processing and analysis," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 963–968.

[30] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: a stream database for network applications," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 647–651.

[31] N. Ashish and C. A. Knoblock, "Semi-automatic wrapper generation for internet information sources," in *Cooperative Information Systems, 1997. COOPIS'97., Proceedings of the Second IFCIS International Conference on*. IEEE, 1997, pp. 160–169.

[32] J. R. Levine, J. R. Levine, T. Mason, and D. Brown, *Lex & yacc*. " O'Reilly Media, Inc.", 1992.

[33] R. Zhang, I. B. Arpinar, and B. Aleman-Meza, "Automatic composition of semantic web services." in *ICWS*, vol. 3, 2003, pp. 38–41.

[34] M. Nagarajan, K. Verma, A. P. Sheth, and J. A. Miller, "Ontology driven data mediation in web services," *International Journal of Web Services Research*, vol. 4, no. 4, p. 104, 2007.

[35] "Vue.js." [Online]. Available: https://vuejs.org/

[36] E. M. Reingold and J. S. Tilford, "Tidier drawings of trees," *IEEE Transactions on software Engineering*, no. 2, pp. 223–228, 1981.

[37] D. Tubbs, "vue-d3-tree-layout," https://github.com/forward-uiuc, 2018.

[38] D. Tubbs, "vue-graphql-input-form-generator," https://github.com/forward-uiuc, 2018.

[39] D. Tubbs, "vue-visual-graphql-query-builder," https://github.com/forward-uiuc, 2018.

APPENDIX

A.1 AN INTROSPECTION QUERY FOR A GRAPHQL SCHEMA

```
query IntrospectionQuery {
  __schema {
    queryType { name }
    mutationType { name }
    subscriptionType { name }
    types {
      ... FullType
    }
    directives {
      name
      description
      args {
        ... InputValue
      }
      onOperation
      onFragment
      onField
    }
  }
 }

fragment FullType on __Type {
  kind
  name
  description
  fields
  (includeDeprecated: true) {
    name
    description
    args {
      ... InputValue
    }
    type {
      ... TypeRef
    }
    isDeprecated
    deprecationReason
  }
  inputFields {
    ... InputValue
  }
  interfaces {
    ... TypeRef
  }
  enumValues(\\
  Deprecated: true) {
  \\
      name
    description
    isDeprecated
    deprecationReason
```

A.1 CONT'D.

```
  }
```

```
    possibleTypes {
      ...TypeRef
    }
  }
    fragment InputValue on
    __InputValue {
      name
      description
      type { ...TypeRef }
      defaultValue
    }

    fragment TypeRef on __Type {
      kind
      name
      ofType {
        kind
        name
        ofType {
          kind
          name
          ofType {
            kind
            name
          }
        }
      }
    }
```