GRAPHITE: A GRAPH SEARCH FRAMEWORK

Sushanta Pradhan

Talentica Software Pvt Ltd., Pune, Maharashtra
sushanta.pradhan@talentica.com

ABSTRACT

With the recent data deluge, search applications are confronted with the complexity of data they handle in terms of volume, velocity and variety. Traditional frameworks such as Lucene [1], index text for efficient searching but do not consider relationships/semantics of data. Any structural change in data demands re-modelling and re-indexing. This paper presents an indexing model that addresses the challenge. Data is modelled as graph in accordance to object-oriented principles such that the system learns the possible queries that can be executed on the indexed data. The model is generic and flexible enough to adapt to the structural changes in data without the need of additional re-modelling & re-indexing. The result is a framework that enables applications to search domain objects, their relationships and related objects using simple APIs without the structural knowledge of underlying data.

KEYWORDS

Graph Search, Semantic Web, Object Oriented Programming, Data Modelling, Algorithms

1. INTRODUCTION

Search is a two-step process:

1. Formulation of a query
2. Execution of the query to extract results.

How does a traditional search application tackle query formulation? Traditional search application left query formulation to the user; it either does not offer or offers little help in this area. Whatever small help it offers, in terms of 'auto text completion' and 'related searches', is based on search patterns of other users. Such help does not work very well as it does not consider semantics of data or user's personality and preferences. In a typical search application users either need to have knowledge of the kind of queries supported or will need to manually filter through the results to extract relevant data. Having the search application prompt the user with relevant queries that can be answered not only lowers the burden on the user but also helps the search application to provide relevant results.

How does a traditional search application tackle extracting results? Traditional search application deploys text based information retrieval techniques to retrieve results and hence only explicit results are returned. For example, first few result pages for the query – 'mobile phone' on popular web search engines, such as Google, Bing, and Yahoo etc. do not have any mention of the inventor of mobile phone – "Martin Cooper". That is to say, search engines just looked at the pages, which contained the text 'mobile phone' but did not consider the relation between 'mobile

Natarajan Meghanathan et al. (Eds) : ICCSEA, SPPR, VLSI, WiMoA, SCAI, CNSA, WeST - 2014
pp. 97–110, 2014. © CS & IT-CSCP 2014 DOI : 10.5121/csit.2014.4718

phone' and 'Martin Cooper' while processing the query – 'mobile phone'. Having search engines consider these relations will enable users to discover more information and gain more knowledge – the most valuable asset in knowledge economy.

With the current data explosion on the Internet, users expect search applications to not only help with query formulation but also provide them relevant information/knowledge that they did not explicitly ask for. Social Graph [2], Knowledge Graph [3], Interest Graph [4], Linked Data [5] are concepts that are currently echoed in the world of Internet. All these concepts focus on the relationships that exist between data and not just the data itself. As applications running on Internet begin to embrace these concepts, it becomes imperative for search applications to also do the same. Graphite is a framework that enables search applications to cater to this demand in today's evolving Internet ecosystem.

The rest of the paper is structured as follows:

Section 2 describes the data model.
Section 3 describes the core components of the system.
Section 4 includes a reference Java implementation.
Section 5 describes the APIs exposed by graphite.
Section 6 briefly describes a graphite backed search application.
Conclusions and related work & future directions are discussed in Section 7 and Section 8 respectively.

2. DATA MODEL

In order to store and make queries based on relationships, we have chosen a 'graph' data structure to model the data. Object Oriented Principles (OOP) are widely used for developing software applications as it closely models the real world and has an inherent graph structure, which we choose to harness. The following entities are modeled in graphite:

1. Domain Class – defined as Class Node
2. Domain Property – defined as Property Node
3. Domain Object/Instance – defined as Object Node

2.1. Class Node (Domain Class)

In an object-oriented environment, application is modeled as classes with certain properties and behavior. An object/instance of such a class, with which the user interacts, is defined as Class Node. For e.g. an online booking system will have objects of type Ticket, Place, Person etc. A blogging application will have objects of type Blog, Author, Topic etc. Graphite looks into the structure of a Class Node to formulate queries that a user might want to ask about objects of this class. Each class node has a unique name and is represented as a node in the underlying store, which has an outward 'name' link pointing to a string. For e.g. class – 'Person' (see figure 1) is represented as shown below:

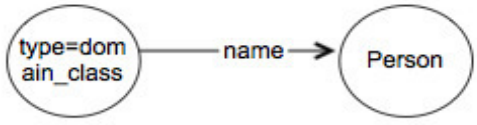

Figure 1: Class Node Structure

2.2. Property Node (Domain Property)

Each property of a class node is defined as Property Node. It has two properties:

1. Range – The type of the property is defined as range.
2. Domain – The class to which the property belongs is defined as domain.

For e.g. Person class (see figure 2) will have properties such as email, location etc.

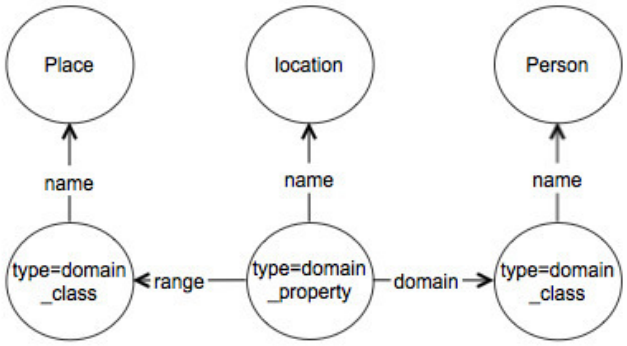

Figure 2: Property Node Structure

2.3. Object Node (Domain Object)

An instance of a class node is defined as Object Node. Each object is represented by a node and has outward property links whose name is same as that of the property name. As shown in figure 3, a domain object named 'ram' has a property named 'location' whose value is 'pune'.

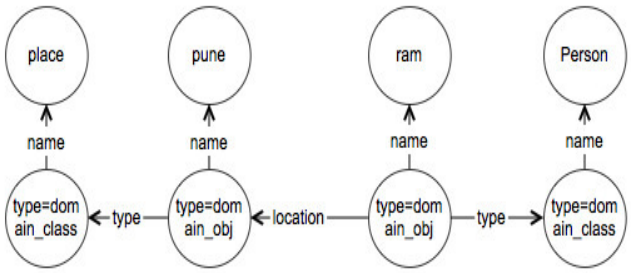

Figure 3: Object Node Structure

3. COMPONENTS

Graphite consists of the following components:

1. Indexer – Stores/Indexes raw data as per the Data Model.
2. Query Advisor – Formulates queries that could be answered by underlying data.

3. Object Searcher – Executes queries formulated by query advisor on indexed data.

4. API – Interface for applications to interact with above three components.

3.1. Indexer

Indexer transforms raw data (classes & instances) to nodes as defined by the data model. Indexing is a two-step process of creating nodes and bonds between them. In the first stage all defined domain classes and their respective properties are added to the index, i.e. Class Nodes and Property Nodes are created. Classes and properties are indexed not just by their name (as is done by a text based indexer) but also by their relationships in the form of 'domain', and 'range' links (as described in Section 2.2). In the second stage instances of an indexed class are added to the index in conformance with the class structure, i.e. Object Nodes are created. Nodes are indexed not just by their names but also by their relationships in the form of 'type' and property links (as described in Section 2.3).

3.2. Query Advisor

This is the gateway to access queries that can be run on the index created by the Indexer. It takes a literal as input, looks into the index and returns possible complete or incomplete [incomplete query isn't syntactically complete to be executed on the indexed data whereas 'Complete Query' can be executed on the indexed data] query objects. Resultant query objects have two types of query embedded in them:

1. Human Readable Query- An English sentence that can be understood by a non-technical end user.

2. Machine Readable Query- A SQL like query that is understood and can be executed by the underlying store, is empty for incomplete queries.

Query Advising is a multi-step process to create possible queries that can be executed on the indexed data. First stage is the seeding stage wherein a literal is fed to the Query Advisor. Here a broad list of relevant complete/incomplete queries related to the inputted literal is returned. In the subsequent stages feedbacks, in the form of expand & contract commands (details discussed in section 4.1.3), from the user is used to complete the selected query.

3.3. Object Searcher

Object Searcher takes a query object as input and returns a list of object nodes that satisfies the given query. It also takes an object node as input and returns its connections, along with the connection strength, for a given level and a given query.

4. IMPLEMENTATION

This section describes a reference Java implementation of graphite.

4.1. Architecture

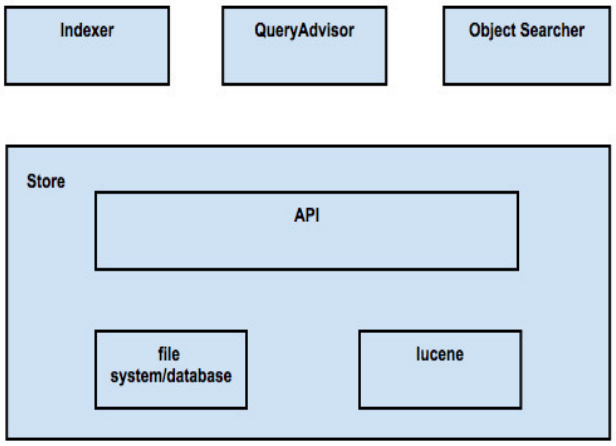

Figure 4: Architecture

4.1.1. Store

This component manages the store and retrieval of nodes. Nodes are persisted either in flat files or database. We have chosen neo4j [6] – a graph database to persist indexed data. Neo4j has support for property graphs, where each node and relationship can have arbitrary properties. To support the ability to have machines interpret data and suggest queries, we have to have a standard and hence we choose RDF [7] graph over property graph. RDF graph has the following advantages:

1. Since RDF is schema less, addition and deletion of indexed data does not demand changes in Graphite application, thus reducing development and maintenance effort for graphite applications.

2. All graphite applications can seamlessly interact and collaborate with each other to produce better search results, thus allowing search application to expand their search horizon without the need for additional integration effort.

3. Object Oriented Principles seamlessly get translated into RDF graph, hence making it easy for the application developers to develop graphite applications.

Each entity (as described in section 2) constitutes a node in the graph and nodes are logically structured into three layers similar to the structuring of carbon nodes in the mineral-'graphite'. Nodes at Layer1 are literals and indexed using Lucene for faster retrieval.

Layer1: The top layer consists of string and number nodes – objects of primitive types.
Layer2: The middle layer consists of class and property nodes.
Layer3: The bottom most layer consists of object nodes.

Nodes of a given layer can be linked (bonded) with another node residing in the same or a different layer. For example, Class definitions can be represented as a graph by considering two classes – 'Book' & 'Author' and their respective instances 'C++' and 'Stroustroup'.

```
class Book{
   Author author;
}
class Author{
   String fullname;
}
```

As seen in the Figure 5 ‘Book’ (#22) and ‘Author’ (#23) are class nodes and their properties ‘author’ (#24) and ‘fullname’ (#21) respectively are property nodes in Layer2. Both the class node and the property node are linked (bonded) by the ‘name’ link to the string node residing in Layer1. Class node is linked by ‘domain’ link to their respective property nodes, for e.g. ‘Book’ (#22) & ‘author’ (#24) are linked and ‘Author’ (#23) & ‘fullname’ (#21) are linked. Property node is linked by ‘range’ link to their respective type (class) node, for e.g. ‘author’ (#24) and ‘Author’ (#23) are linked. Object nodes ‘C++’ (#31) and ‘Stroustrup’ (#32) reside in Layer3. These object nodes are linked to nodes, in Layer1 by name link, in Layer2 by type link, in Layer3 by property links. All links except ‘name’ and ‘type’ are created in adherence with the links of class node (instance type) and property nodes in Layer2.

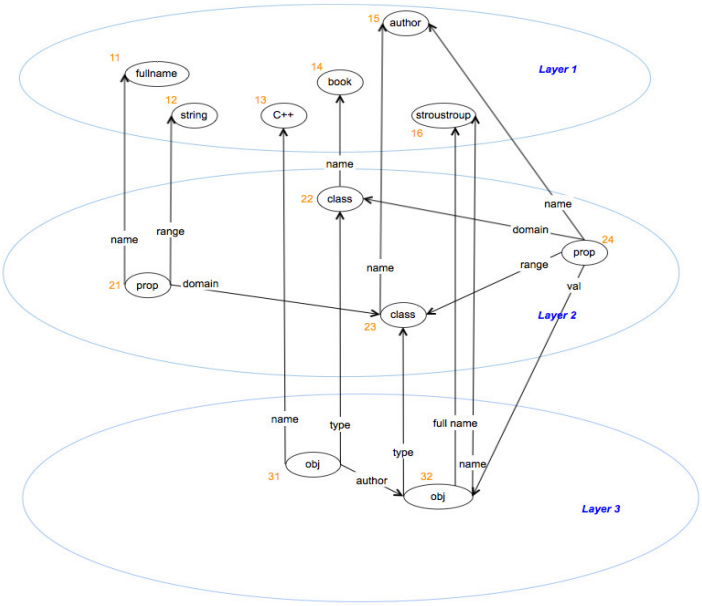

Figure 5: Data Model

4.1.2. Indexer

This is the gateway to write nodes to store; a graphite application interacts with this component to persist nodes as per the data model described in Section 2. It exposes APIs (discussed in section 5) to define domain classes, domain properties and domain objects and add them to store.

4.1.3. Query Advisor

Query Advisor manages the formulation of queries starting from a keyword and subsequent commands: expand or contract. Since the starting point is a keyword, it starts from a node of Layer1 and follows the name link to fetch the first node of query stack (described below). Node at

Layer1 is linked (bonded) to either class, property node of Layer2 or object node of Layer3, hence Query Advisor produces two types of queries based on type of node it finds.

Diverging Query: Following the name link, if the query advisor lands on Layer3, it produces a diverging query. This type of query is called diverging query because the graph structure that gets outputted after execution is of diverging nature as seen in figure 6. The output nodes for such queries are the nodes at level 1.

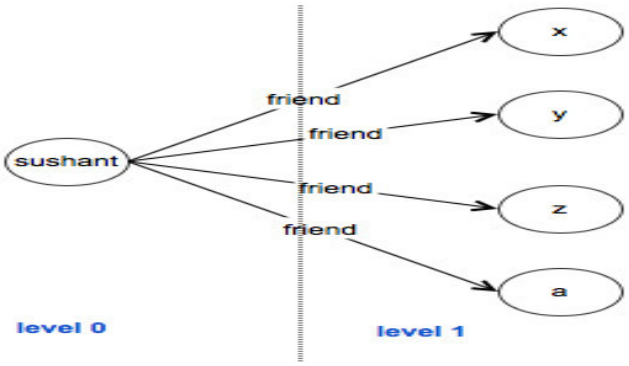

Figure 6: Diverging Query Result Structure

Converging Query: Following the name link, if the query advisor lands on Layer2, it produces a converging query. The graph structure that gets outputted is converging in nature as seen in figure 7 and hence the name. Nodes of level 0 are outputted for such queries.

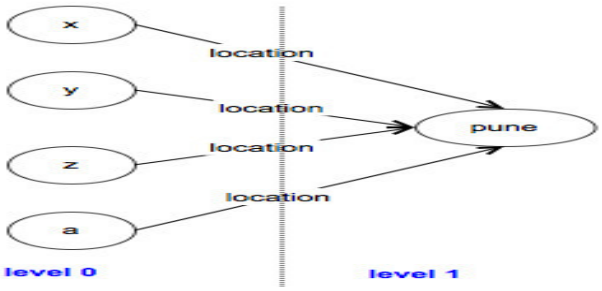

Figure 7: Converging Query Result Structure

Query Stack: It is a collection of nodes stacked one above another in accordance to the structure of class nodes residing in Layer2. It is initialized by name of either a class node or an object node, which becomes the first element of the stack. After a query chain is initialized which can be either expanded or contracted. On expansion, Query Advisor checks the type of node at top of stack and performs the following actions.

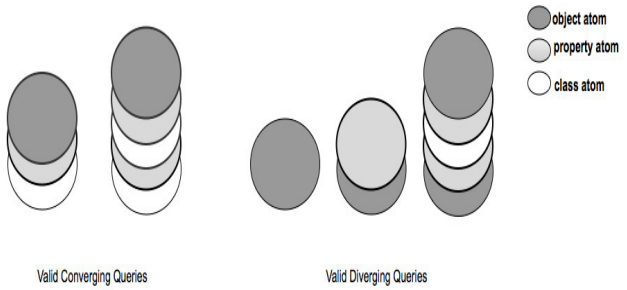

Figure 8: Query Stacks

If the top node is found to be a class node, Query Advisor follows the domain link and fetches all property nodes linked to the class node. Since there could be more than one property node, it creates as many copies of the original Query Stack as the number of property nodes found and pushes one property node per copy. Finally this list of Query Stack is returned.

If the top node is found to be an object node, Query Advisor follows the type link and fetches the class node from Layer2. It assumes this class node to be at the top of query stack and performs the expansion as explained above.

If the top node is found to be a property node, Query Advisor follows the 'val' link to find the linked object nodes at Layer3 and range link to find linked class nodes. Since there could be more than one node found, it creates as many copies of original Query Stack as the number of nodes found and pushes one node per copy. Finally the list of Query Stack is returned. Contraction of a query stack simply removes the top node.

Human Readable Query: Human Readable Query is translation of Query Stack to an English sentence by the Query Advisor that a non-technical user can understand. A property of a class can be considered as possessive adjective [8] for the noun [9] - class and hence the class, property and its value can be translated into an English language sentence as *"<class_name> whose <property_name> is <property_value>*". Using this principle a Query Stack of Diverging Query is translated into perceivable human readable query. In case of a Converging Query, the sentence is of the form *"<object_name>'s <property_name>*". A Human Readable Query can be generated for a Query Stack irrespective of its current state.

Machine Readable Query: Similar to Human Readable Query, Machine Readable Query is a translation of a Query Stack into a form that is executable by Object Searcher. As nodes in the query stack are stacked in accordance with links that exist between them, they are parsed to form cypher [10] query. For e.g. to find 'all books authored by 'Stroutstrup' the cypher query would be (see Converging Query of figure 9) - "*start book_class = node(22), author = node(32) match (book_class)<-[:type]-(book)-[:author]-(author) return book;*". Similarly the author of the book named 'C++' the cypher query (see Diverging query of figure 8) would be - "start book = node(31) match (book)-[:author]->(authors) return author;". Unlike Human Readable Queries, Machine Readable Query is generated only when Query Stack is in complete state.

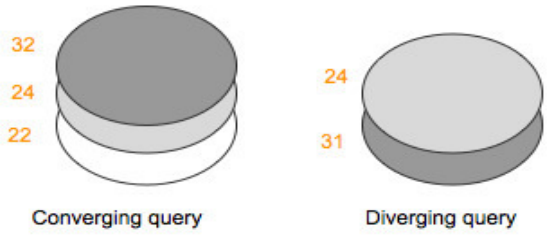

Figure 9: Query Stack Example

4.1.4. ObjectSearcher

Object searcher is primarily responsible for searching nodes of Layer3 that satisfies a given query. It does so by executing the Machine Readable Query extracted from the corresponding Query Stack. Two lists are returned when a Machine Readable Query gets executed:

1. List of ids corresponding to object nodes that satisfy the given query. For e.g. for the converging query (of Figure 9), the list: [31,] is returned.
2. List of related class node ids. For e.g. for the converging query (of Figure 9), the list: [23,] is returned.

Object Searcher also extracts connections of a given object node along with their routes and strength. Strength (S_c) of a connection gives a measure of how strongly two objects are connected to each other. It is given by the formula:

$$S_{(c,o)} = \sum_r S^r_{(c,o)}$$

$S_{(c,o)}$ – Srength of a connection (c) of object node (o).

$S^r_{(c,o)}$ – Strength of a route (r) of a connection (c) of object node (o). Route is simply the edges and nodes that are traversed to reach the connection from the object node. Strength of a route is given by the formula:

$$S^r = \frac{\sum_a R_a}{L_r^2}$$

R_a – Rank of node (a).
L_r – Path length of route (r).

5. API

This is an interface for applications to interact with Graphite i.e. to index data and retrieve it back. Annotations are used to annotate domain classes whose instances are to be indexed. Indexer and Search APIs are used to index and retrieve data respectively.

5.1 Annotations

@ClassNode: A class level annotation that is used to define a Domain Class. Its arguments define 'name' and 'rank' of the annotated java class. Default value of 'name' is the name of the class itself and default value for 'rank' is 1. The rank value is used for ranking the search results, which is described in detail in section 4.1.4.

@PropertyNode: A property level annotation used to annotate class properties. Its arguments define 'name' and 'rank' of the annotated property.

Writing a Graphite program involves the following steps:

1. Indexing
2. Searching

5.2. Indexing

At this stage, domain classes are defined along with their relations to other domain classes and added to the index. Thereafter objects/instances of domain class are added to the index.

5.2.1. Adding a domain class to the index

Domain class is defined by sub classing the predefined 'ObjectNode' class and annotating it with @ClassNode annotation (see Figure 10). Index-able properties are annotated with the @PropertyNode annotation. The Class- GraphiteAnnotationParser searches a given package and adds the domain classes along with their properties to the index (see Figure 10).

```
@ClassNode(rank=2)
public class Author extends ObjectNode{
        public Author(String name) {
                super(name);
        }
        @PropertyNode(name="author", rank = 2)
        private String fullname;
}
```

Figure 10: Class Node Definition

5.2.3. Adding a domain object to the index

ObjectNodeWriter.write(object) API is called to add a domain object to the index (see Figure 11). Only instances of a domain class can be indexed.

5.3. Searching

At this stage user makes a query to get valid queries and passes one of these queries to the search API to fetch domain objects honoring the given query. It is a three-step process:

1. Querying for queries
2. Querying for objects
3. Querying for connections

5.3.1. Querying for queries

The kind of queries that can be run on a given set of data is best known by the data itself. Graphite inspects the indexed data it handles and suggests queries to the user. QuerySuggestor.getQuery(keyword) API is used to get the list of possible queries (see Figure 11).

```
StoreResources sr = new StoreResources ("/tmp", "/tmp/db.prop");
GraphiteAnnotationParser gap = new
GraphiteAnnotationParser("com.talentica.graphite.domain", sr);
//add class nodes to index
gap.parse();
ObjectNodeWriter writer = new ObjectNodeWriter(sr);
Author author = new Author("Stroustroup");
//add object node to index
writer.write(author);
QuerySuggestor qs = new QuerySuggestor(sr);
//get list of queries for the keyword 'author'
List<Query> queries = qs.getQuery("author");
ObjectSearcher searcher = new ObjectSearcher(sr);
//search object nodes for a query
List<ObjectNode> result = searcher.search(queries.get(0));
```

Figure 11: Index and search example

5.3.2. Querying for objects

At this stage the API- ObjectSearcher.search (query) is called with a query to fetch object nodes honoring the given query (see Figure 11).

5.3.3. Querying for connections

ObjectSearcher.getConnections (level, objectId, query) API is called to retrieve the related object nodes for a given object node. The ‘level’ argument defines the depth to which related objects are to be searched, ‘objectId’ is the id of the object node whose connections will be searched for and the ‘query’ argument defines the criteria to be honored by the related object nodes.

6. SEARCH APPLICATION

Graphite is primarily a graph search tool; hence search becomes the primary application of Graphite. By using graphite as a backend, search can be enhanced from text-based to object-based; results that were not explicitly queried for can be outputted. For e.g. search with keyword – ‘book’, outputs all books in the ‘primary pane’, while related objects of type - ‘author’ & ‘publisher’ are displayed in ‘connection pane’ (see figure 12). Clicking or hovering over a connection shows the path that was traversed to reach to the connection from the object in the primary pane (arrows in blue in the below figure).

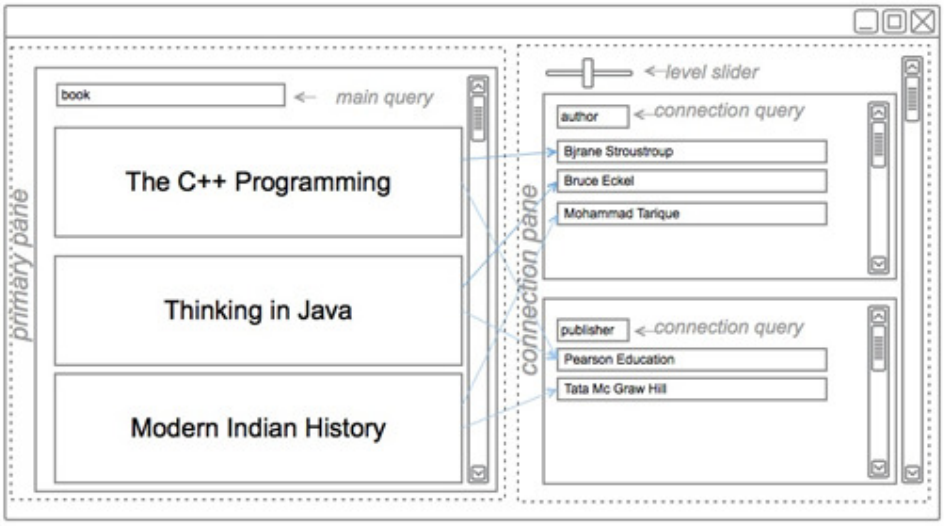

Figure 12: UI of a search application backed by graphite

By modifying a connection query, user can filter/expand his main query results (primary pane). For e.g. if the connection query 'author' is changed to 'Bjarne Stroustrup', elements – 'Thinking in Java' and 'Modern Indian History' would disappear from main query results, and 'Bruce Eckel', 'Mohammad Tarique' and 'Tata Mc Graw Hill' would disappear from connection pane.

By changing the level slider, users would be able to discover more facts/knowledge related to main query results, which in turn would lead to enhanced user satisfaction.

7. RELATED & FUTURE WORK

To our knowledge, there isn't any open-source framework/library similar to Graphite, though there are few proprietary implementations such as Facebook's Graph Search [12] and Google's Knowledge Graph. Facebook's Graph Search, is similar to a Graphite application as it is also object-based, has a query advisor and outputs related objects. Google's Knowledge Graph shows related objects, provides user with results that were not explicitly asked by the user.

We have implemented the basic features required by a search application; issues related to security, scalability has not been analyzed. Future releases of Graphite would focus to address these issues.

Currently, our core focus and inspiration of Graphite was search. Hence we have only concentrated on 'nouns' and 'adjectives'; therefore the data model focuses only on that part of OOP principles. Also, the two common graph models - RDF graph and property graph considers only 'nouns' and 'adjectives'. It would be an interesting research foray to also model 'verbs' as it has a possibility to transform a SEARCH-ENGINE into a DO-ENGINE.

8. CONCLUSIONS

Graphite is an indexing framework, focused on providing users with a natural API for indexing classes, properties, objects and their relationships. In concept, it is similar to Lucene but with a focus on relations.

Traditionally, search applications were built to provide local search to local users i.e. they handled only application data and served only application users. Since users of the application had fair know-how of data provided by the application, users could formulate queries to suffice their needs. From the software developer perspective, maintaining a text-based inverted index using Lucene sufficed to cater such needs of user. Data deluge, increasing use of data by modern day applications (mashup [11]), dynamic nature of data and increasing sources of open data poses problems at the supply side as well as at the demand side.

At demand side, users do (can) not have (gain) fair know-how of data and hence can't formulate efficient queries for search engines to cater their demand efficiently. Graphite's QueryAdvisor helps the user at run-time to gain know-how about data as it can prompt user with valid queries. Graphite's ability to output related objects further enhances user's know-how about data.

At supply side, developers have to constantly make changes to adapt to the changing nature of application data. Graphite's ability to understand OOP principles, developers can write hooks to index searchable data from the core application itself and need not write/modify explicit code for search application.

With increasing sources of open data, users expect search applications to give results from such sources as well and not just the local data. Since Graphite data model embraces the principles of OOPs – a universally used principle to write web applications, graphite applications can collaborate with each other to provide users a rich search experience.

ACKNOWLEDGEMENTS

I would like to thank my mangers – Aniket Shaligram and Manjusha Madabushi for providing access to a live product's data for conducting experiments and their valuable comments to improve the model.

REFERENCES

[1] Michael MacCandless, Erik Hatcher, Otis Gospodnetić. (2010). Lucene in Action
[2] Social Graph web page: http://en.wikipedia.org/wiki/Social_graph
[3] Knowledge Graph Web page: http://en.wikipedia.org/wiki/Knowledge_Graph, http://www.google.co.in/insidesearch/features/search/knowledge.html
[4] Interest Graph Webpage: http://en.wikipedia.org/wiki/Interest_graph
[5] Linked Data Standards webpage: http://www.w3.org/standards/semanticweb/data
[6] Jonas Partner, Aleksa Vukotic, and Nicki Watt. (2012). Neo4j in Action.
[7] RDF Working Group, "Resource Description Framework (RDF): Concepts and Abstract Syntax", Klyne G., Carroll J. (Editors), W3C Recommendation 10 February 2004
[8] Possessive Adjective definition: http://en.wiktionary.org/wiki/possessive_adjective
[9] Noun Definition: http://en.wiktionary.org/wiki/noun
[10] Cypher Query Language webpage: http://docs.neo4j.org/chunked/stable/cypher-query-lang.html
[11] Mashup webpage: http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)
[12] Facebook Graph Search webpage: https://www.facebook.com/notes/facebook-engineering/under-the-hood-building-graph-search-beta/10151240856103920

AUTHOR

Sushanta Pradhan has around 7 years of product development experience mostly in JAVA/J2EE technologies. His interests include semantic web, machine learning, parallel/distributed computing and JVM languages. At Talentica he has architected and built highly scalable and maintainable applications in the digital advertisement and marketing space. He has done his B.Tech from NIT Calicut in Electronics and Communication Engineering.