A REPORT
ON

Graph Data Digest Document Format (GDF)

BY

Vipin Baswan	2017A7PS0429P
Suyash Raj	2017A7PS0191P
Yashdeep Gupta	2017A7PS0114P
Abhinava Arsada	2017A7PS0028P
Sreyas Ravichandran	2017A7PS0275P

AT

Homi Bhabha Centre for Science Education (HBCSE)

A Practice School-1 station of

Birla Institute of Technology & Science, Pilani

(June 2019)

A REPORT
ON

Graph Data Digest Document Format (GDF)

BY

Vipin Baswan	2017A7PS0429P	Computer Science
Suyash Raj	2017A7PS0191P	Computer Science
Yashdeep Gupta	2017A7PS0114P	Computer Science
Abhinava Arsada	2017A7PS0028P	Computer Science
Sreyas Ravichandran	2017A7PS0275P	Computer Science

Prepared in the partial fulfilment of the

Practice School-I (BITS F221)

AT

Homi Bhabha Centre for Science Education (HBCSE)

A Practice School-1 station of

Birla Institute of Technology & Science, Pilani

(June 2019)

ACKNOWLEDGEMENTS

We would like to thank **Dr K Subramaniam**, Director, **Homi Bhabha Centre for Science Education** for providing this opportunity to us, the students of BITS Pilani, to work with the organization towards the fulfilment of its purpose of promoting science education throughout the country. We'd also like to thank **Nagarjuna G.** to allow us the chance to work on this project under his able guidance. We also extend our thanks to **Prof. Mukesh Kumar Rohil**, our Practice School Coordinator for his constant support and guidance. Additionally, we express our heartfelt gratitude to **Mr. J. B. Waghmare** for his constant support at the PS station in non-academic matters.

Finally, our humblest apologies to all others who helped us but whose names could not be mentioned in the list.

Birla Institute of Technology and Science

Pilani (Rajasthan)

Practice School Division

Station: Homi Bhabha Centre for Science Education **Centre:** Mumbai

Duration: 21 days (till the submission of the Mid-Semester report)

Date of Start: 21st May, 2019

Date of Submission: 10th June, 2019

Title of Project: Graph Data Digest Document Format (GDF)

Submitted By:

Vipin Baswan	2017A7PS0429P	Computer Science
Suyash Raj	2017A7PS0191P	Computer Science
Yashdeep Gupta	2017A7PS0114P	Computer Science
Abhinava Arsada	2017A7PS0028P	Computer Science
Sreyas Ravichandran	2017A7PS0275P	Computer Science

Name(s) of expert(s):

Nagarjuna G., faculty at Homi Bhabha Centre for Science Education

Name of the PS Faculty:

Prof. Mukesh Kumar Rohil

Key Words: GDF, Graph Databases, RDF, GraphQL, Data Digest Format, SPARQL, Renderer, 7-column format, MetaData

Project Areas: Graph Databases and querying languages

Abstract

Graph databases have always been a promising tool in increasing the querying efficiency on datasets. Hence, the prospect of data digest document format such as GDF seems very promising in today's world where datasets interact in a complex manner and quick information retrieval is of prime import.

Our project deals with developing a format called GDF and the method to convert any document format into GDF. This will assist us in quick merging of different files as graphs can be merged easily. On completion of our project, we will be able to convert any file format into GDF and also view any file data in the form of graphs (only nodes and edges)! This format is essentially NoSQL type format since there are no tables.

_____	_____
Signature of the Student	Signature of PS Faculty
_____	_____
Date	Date

TABLE OF CONTENTS

INTRODUCTION

The aim of our project is to develop a Data Digest Format which can be used to convert and represent information of any format. The format is graph based, hence the name.

The scope of our project is to:

a) Decide the format of GDF
b) Write methods to convert a text file into GDF
c) Create meta-data in the GDF format from the meta-data in the text format
d) Develop a query language (based on SPARQL) for information retrieval

Since the idea of GDF is pretty innovative and unique in itself (Credit: Nagarjuna G.), not much literature is available to us for this exact format. But a similar data format called *Resource Description Format* (RDF) already exists. It is also graph based data format. Hence, we have gone through the literatures regarding RDF (links of the online resources have been given in the References section). Also, we have generated the sample data (for testing our code) using a python code but later we will collect more data from *DBPedia* (an online platform to get 3-column formatted data for various Wikipedia pages). Also, the query language for our format is based on *SPARQL*. We are referring to the official literature available on *SPARQL* (link for the same has been provided in References Section) for building our querying engine.

We wanted to limit the dependency of our code on various platforms. Hence, we have used BASH Scripting to write our code.

Due to time constraints, we will not be able to create our own renderer but we will be using already existing D3.js renderer.

The report initially gives a basic background information on Graph based database formats. Then, the *7-column format* has been discussed in detail. Since, the format is graph based, we also need to define nodes and edges of the graph. This is done through generating the meta-data file. The format of meta-data file is also discussed in detail. After this, the report explains the flow of the project (from text to Graph) through a flow chart. Since we have limited time to complete the project, we have also stated the current progress and what we have to achieve in two separate sections. The relevant code and readMe files have been attached in the appendix. The links of various resources we referred to have been given in the references section. Few important jargons have been defined in the glossary at the end.

1. BACKGROUND

This data format can be said to be loosely inspired from the RDF format, additionally making use of the seven-column format, which has been described below. In both of these formats, *data is stored in the form of graphs* i.e. nodes as well as edges for easier and more efficient querying of data. We also decided to additionally generate unique IDs for each one of the tuples generated as well as each entity uniquely specified by the edges and nodes. The last addition to this data format is that we shall implement a constantly self-updating metadata section of our data which cannot be accessed by the users and contains information about the type of entities stored in our data which reduces our query time to a very large extent despite requiring a very large amount of storage space.

Tools used in this project are:

a) **Shell Scripting:** *to write most of the code*
b) **RDF:** *to understand how text can be shown as graph*
c) **D3.js:** *Renderer to get graph SGV*
d) **SPARQL:** *to design querying engine for GDF*

2. GRAPH DATABASES

Graph Databases embrace relationships. Sometimes we want to view the entities along with the various relationships between them.

Graph databases have huge applications. Now comes the question, *why do we need graph databases at all?*

a) Better Performance: Since there are no joins, recursive queries are localized to only a part of graph.
b) Flexibility: New relationships (i.e. subgraphs) can be added without disturbing existing database structure and queries, since schema is not fixed.
c) Maintenance: Application interface can be modified without any change to data definition schema.

Fig. 1 shows an example of a social network graph:

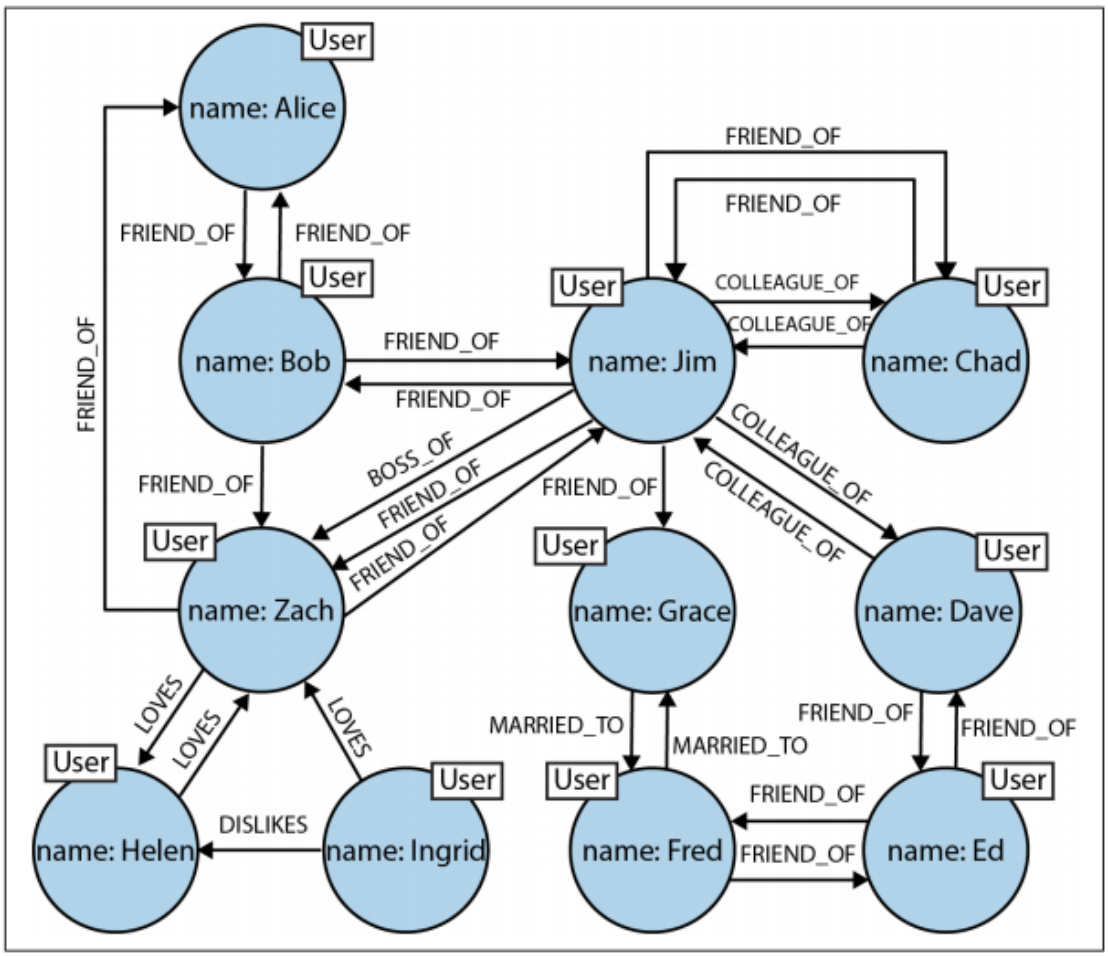

Fig. 1: Modelling friends, colleagues, workers, and (unrequited) lovers in a graph

A social network is a popular example of a densely connected, variably-structured network, one that resists being captured by a one-size-fits-all schema. *The flexibility of the graph model allows us to add new nodes and new relationships without compromising the existing network or migrating data—the original data and its intent remain intact.*

The graph offers a much richer picture of the network. We can see who LOVES whom (and whether that love is requited). We can see who is a COLLEAGUE_OF whom, and who is BOSS_OF them all. We can see who is MARRIED_TO someone else; we can even see the antisocial elements in our otherwise social network, as represented by DISLIKES relationships.

3. RESOURCE DESCRIPTION FRAMEWORK (RDF)

Concepts and Abstract Syntax

The Resource Description Framework (RDF) is a framework for representing information in the Web. The framework is designed so that vocabularies can be layered. The RDF and its vocabulary definition (schema) languages are the first such vocabularies.

The design of RDF is intended to meet the following goals:

a) having a simple data model
b) having formal semantics and provable inference
c) using an extensible URI-based vocabulary
d) using an XML-based syntax
e) supporting use of XML schema datatypes
f) allowing anyone to make statements about any resource

Graph Data Model

The underlying structure of any expression in RDF is a collection of triples, each consisting of a subject, a predicate and an object. A set of such triples is called an RDF graph. This is illustrated in Fig. 2 by a node and directed-arc diagram:

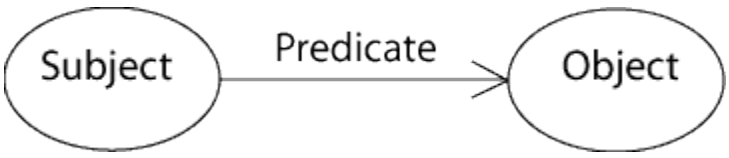

Fig. 2: The triples in the form of the graph

Each triple represents a statement of a relationship between the things denoted by the nodes that it links. Each triple has three parts:

1) the subject, which is an RDF URI reference or a blank node
2) the predicate, which is an RDF URI reference
3) the object, which is an RDF URI reference, a literal or a blank node

RDF triple is conventionally written in the order subject, predicate, object. The direction of the arc is significant: it always points toward the object. The nodes of an RDF graph are its subjects and objects.

The assertion of an RDF triple says that some relationship, indicated by the predicate, holds between the things denoted by subject and object of the triple. The assertion of an RDF graph amounts to asserting all the triples in it, so the meaning of an RDF graph is the conjunction (logical AND) of the statements corresponding to all the triples it contains.

RDF Expression of Simple Facts

Some simple facts indicate a relationship between two things. Such a fact may be represented as an RDF triple in which the predicate names the relationship, and the subject and object denote the two things. A familiar representation of such a fact might be as a row in a table in a relational database. *The table has two columns, corresponding to the subject and the object of the RDF triple. The name of the table corresponds to the predicate of the RDF triplet.*

Relational databases permit a table to have an arbitrary number of columns, a row of which expresses information corresponding to a predicate in first order logic with an arbitrary number of places. Such a row, or predicate, has to be decomposed for representation as RDF triples. A simple form of decomposition introduces a new *blank node*, corresponding to the row, and a new triple is introduced for each cell in the row. The subject of each triple is the new blank node, the predicate corresponds to the column name, and object corresponds to the value in the cell. The new blank node may also have an **`rdf:type`** property whose value corresponds to the table name.

Fig. 3 demonstrates an example of the above decomposition:

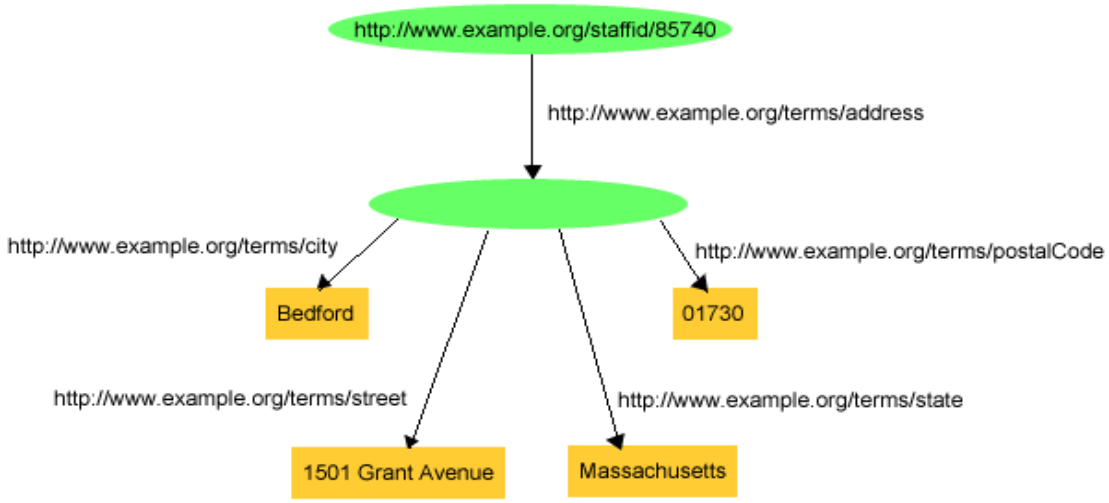

Fig. 3: Using a Blank Node

This information might correspond to a row in a table **"STAFFADDRESSES"**, with a primary key **STAFFID**, and additional columns **STREET**, **STATE**, **CITY** and **POSTALCODE**.

Thus, a more complex fact is expressed in RDF using a conjunction (logical-AND) of simple binary relationships. RDF does not provide means to express negation (NOT) or disjunction (OR).

Through its use of extensible URI-based vocabularies, RDF provides for expression of facts about arbitrary. A URI can be constructed for anything that can be named, so RDF facts can be about any such things.

Entailment

The ideas on meaning and inference in RDF are underpinned by the formal concept of entailment. *In brief, an RDF expression A is said to entail another RDF expression B if every possible arrangement of things in the world that makes A true also makes B true*.

Graph Equivalence

Two RDF graphs *G* and *G'* are equivalent if there is a bijection *M* between the sets of nodes of the two graphs, such that:

1) *M* maps blank nodes to blank nodes.
2) *M(lit)=lit* for all RDF literals *lit* which are nodes of *G*.
3) *M(uri)=uri* for all RDF URI References *uri* which are nodes of *G*.
4) The triple *(s, p, o)* is in *G* if and only if the triple *(M(s), p, M(o))* is in *G'*

RDF URI References

A URI reference within an RDF graph (an RDF URI reference) is a Unicode string that:

1) does not contain any control characters (#x00 - #x1F, #x7F-#x9F)
2) and would produce a valid URI character representing an absolute URI with optional fragment identifier when subjected to the encoding described below.

4. 7-COLUMN FORMAT

Each tuple in the input file (can be in any format) can be viewed as an entity with a Subject, Object and a Predicate. For instance, in "*Yashdeep likes to eat ice-cream*", 'Yashdeep' is the Subject, 'ice-cream' is the Object and 'likes to eat' is the Predicate.
Let's see the graph given below:

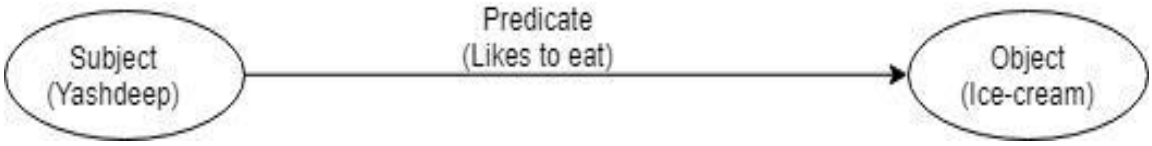

Fig. 4: A graph representing the sentence "Yashdeep like to eat ice-cream"

Here are the observations:

a) Both subject and object are represented by the nodes of the graph
b) Predicate is represented by the edge of the graph
c) Both nodes and edges have some text associated with them (like 'Yashdeep' and 'Ice-cream' associated with nodes and 'Likes to eat' associated with edge)
d) The edge originates from Subject and terminates at Object. Thus, *the graph we get is always a directed graph*
e) Subject, Objects and Predicates can also have '*qualifiers*' associated with them. For example, in above graph, Subject Qualifier can be 'Person', Object Qualifier can be 'Dessert' and Predicate Qualifier can be 'Preferences'. *In short, a qualifier gives more information about the subject/object/predicate.*

The above graph corresponds to a single tuple of the GDF file. This graph is represented by a 7-column format in our GDF File. The format is:

UID | Subject | Subject_Qualifier | Predicate | Predicate_Qualifier | Object | Object_Qualifier

Hence, above graph will be represented by the following tuple in our GDF File:

1242353353|Yashdeep|Person|LikesToEat|Preferences|Ice-cream|Dessert

The UID is generated by taking the hash of Subject, Predicate and Object (appended and then delimited by space). We have used built-in 'md5sum' hash of the bash. For each tuple in the input file, a tuple is generated in the GDF file. Finally, the output will contain a file with .gdf extension.

The code for the conversion of text into GDF file is attached in the appendices section along with the ReadMe file for the code. For complete detailed

explanation on how the code works and what 7-column format is, please refer to the ReadMe file.

5. META-DATA FORMAT

Along with the GDF file a meta-data GDF file is also created. For creating this file, the user has to give the meta-data text file as the input. The meta-data file will also have a Graph based file format. For instance, see the graph below:

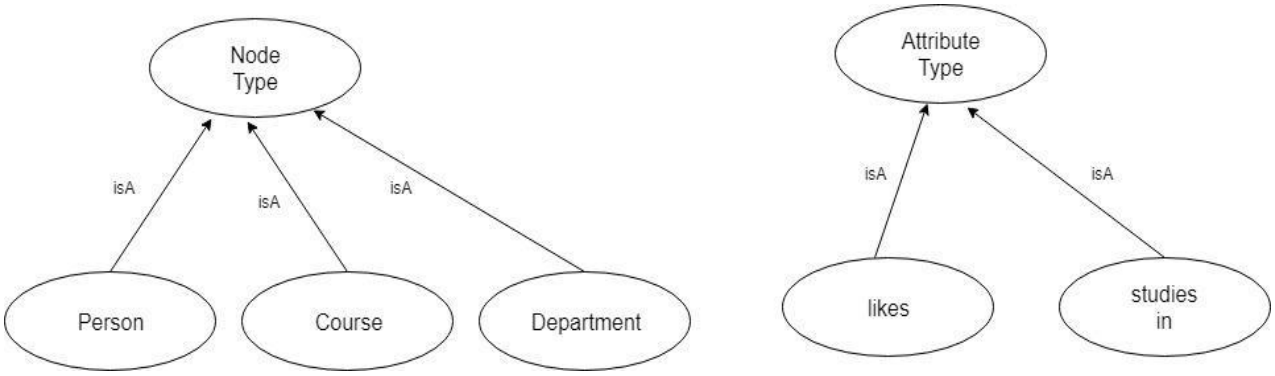

Fig. 5: An example of Node and Attribute types for meta-data.gdf file

The meta-data contains two basic entities: Node type and Attribute/Predicate type. This is used to define the nodes and edges within the graph.

In the example discussed previously, "*Yashdeep likes to eat ice-cream*", Yashdeep is a Person and hence Yashdeep is a Member of Person Node type. This information can also be conveyed through the 7-column format:

UID | Subject | Subject_Qualifier | memberOf | | Node_Type |

UID | Predicate | Predicate_Qualifier | memberOf | | Attribute_Type |

UID | Object | Object_Qualifier | memberOf | | Node_Type |

In our example,

1243453453 | Yashdeep | | memberOf | | Person |

1453456564 | Ice-cream | | memberOf | | Food-items |

2434536456 | Likes to eat | | memberOf | | likes |

Hence, for each subject/object/predicate, our meta-data.gdf file will contain a tuple in it.

6. TEXT TO GRAPH CONVERSION

The conversion of input text file into Graph database can be represented by the following chart:

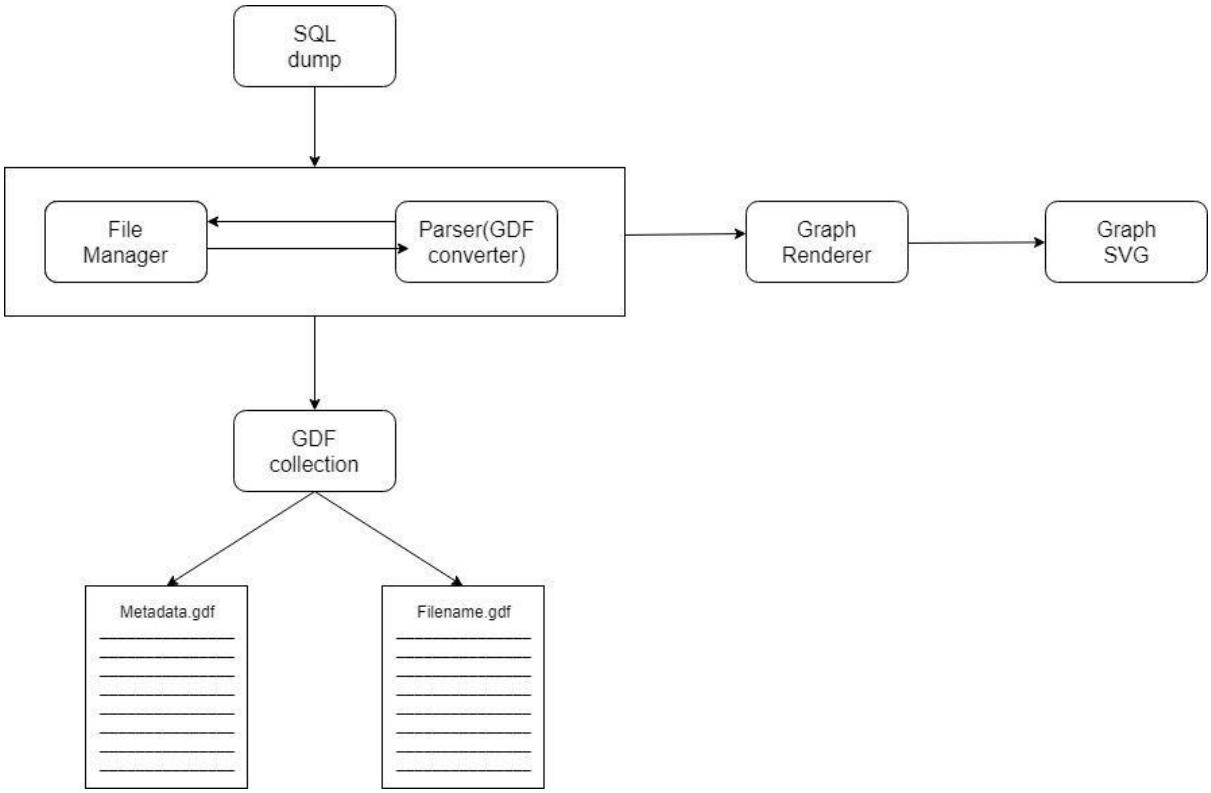

Fig. 6: Processing of input file to get an interactive Graph SVG

The sequence of steps has been explained below:

a) We will receive input from the user in the form of a table (called SQL Dump in the above chart). Mostly, user will also provide the corresponding meta-data file as another input file.
b) The parser will parse the input file and generate the GDF file along with the meta-data.gdf file. The availability of meta-data.gdf rests upon the availability of meta-data file as the input from user.
c) We need to convert the GDF file into the JSON objects and then render these JSON objects to Graph SVG. This will be done by the Graph Renderer. It will read our GDF file and generate the corresponding graph database.

7. ABOUT SPARQL

We are using SPARQL protocol in order to create a query engine. SPARQL allows users to write queries against what can loosely be called "key-value" data or, more specifically, data that follow the RDF specification of the W3C (World Wide Web Consortium). Thus, the entire database is a set of "subject-

predicate-object" triples. This is analogous to some NoSQL databases' usage of the term "document-key-value", such as MongoDB.

In SQL relational database terms, RDF data can also be considered a table with three columns – the subject column, the predicate column, and the object column. The subject in RDF is analogous to an entity in a SQL database, where the data elements (or fields) for a given business object are placed in multiple columns, sometimes spread across more than one table, and identified by a unique key. In RDF, those fields are instead represented as separate predicate/object rows sharing the same subject, often the same unique key, with the predicate being analogous to the column name and the object the actual data.

Unlike relational databases, the object column is heterogeneous: the per-cell data type is usually implied (or specified in the ontology) by the predicate value. Also, unlike SQL, RDF can have multiple entries per predicate; for instance, one could have multiple "child" entries for a single "person", and can return collections of such objects, like "children".

Thus, SPARQL provides a full set of analytic query operations such as JOIN, SORT, AGGREGATE for data whose schema is intrinsically part of the data rather than requiring a separate schema definition. However, schema information (the ontology) is often provided externally, to allow joining of different datasets unambiguously. In addition, SPARQL provides specific graph traversal syntax for data that can be thought of as a graph.

Query Forms

In the case of queries that read data from the database, the SPARQL language specifies four different query variations for different purposes.

a) **SELECT query:**
   Used to extract raw values from a SPARQL endpoint, the results are returned in a table format.

b) **CONSTRUCT query:**
   Used to extract information from the SPARQL endpoint and transform the results into valid RDF.

c) **ASK query:**

Used to provide a simple True/False result for a query on a SPARQL endpoint.

d) DESCRIBE query:
Used to extract an RDF graph from the SPARQL endpoint, the content of which is left to the endpoint to decide based on what the maintainer deems as useful information.

NOTE: Each of these query forms takes a WHERE block to restrict the query, although, in the case of the DESCRIBE query, the WHERE is optional.

MISCELLANEOUS

Variables are indicated by a ? or $ prefix. When a triple ends with a semicolon, the subject from this triple will implicitly complete the following pair to an entire triple. So, for example ***ex:isCapitalOf ?y*** is short for ***?x ex:isCapitalOf ?y***.
The SPARQL query processor will search for sets of triples that match these four triple patterns, binding the variables in the query to the corresponding parts of each triple. To make queries concise, SPARQL allows the definition of prefixes and base URIs in a fashion similar to Turtle. In every query, the PREFIX "rdfs" stands for <http://www.w3.org/2000/01/rdf-schema#>.

8. OUR PROGRESS

Till now, we have completed the following:

a) Creation of GDF Convertor to convert a text file into 7-column format
b) Creation of meta-data.gdf file from meta-data.txt file
c) Reading SPARQL

We will also start working on the querying engine within this week.

Everything that we have done has been committed to the GIT remote repository (https://github.com/Sreyas-108/GDF). Also, the code and readMe files have been attached in the Appendix section.

9. FUTURE PROSPECTS

We plan to complete the following objectives:

a) Conversion of our GDF file into JSON objects
b) Understand the working of D3.js renderer and use it to generate graph SVGs from the JSON objects generated from (a)
c) Complete the building of querying engine (based on SPARQL)

10. SKILLS ACQUIRED

We have learnt the following:

a) Bash
b) Shell Scripting
c) RDF
d) SPARQL
e) JSON
f) GitHub

CONCLUSION

Graph databases show enormous promise in terms of efficiency, by one or more orders of magnitude and with latency much lower compared to batch processing of aggregates. Additionally, graph databases have a very flexible data model and a mode of delivery conforming to modern methods. GDF is another innovation in the direction. The aim is to create and modify a graph data digest and build a database engine to Create, Read, Update and Delete (CRUD). There are many steps in the process, the underlying storage, a GDF parser, a querying engine and a renderer. The paucity of time may not allow us to finish what we've started but we hope to build a foundation on which further progress can be made.

APPENDIX

NOTE: The code attached here is already present on GitHub (https://github.com/Sreyas-108/GDF). Also, for sample input data and corresponding GDF file, please refer to the GitHub link.

Python code to generate sample input data:

```
import random
#Create some lists with dummy data
students=['Yashdeep','Vipin','Suyash','Sreyas','Abhinava','Ayush','S
id','Anirudh','Arvind','Ravi','Kavi','Isha','Ritu','Jaya','Kapil','D
ivya','Pankaj','Sashan','Sushakt','Pratik','Saksham','Rachit','Shiva
m','Shalvi','Akriti','Bhoomi','Kavya','Sahil','Sargun','Swadesh','Ks
hitij','Anshuman','Samir']
teachers=['Ashish','Shan','Geeta','Ashu','Nago','Pogo','Sanjay','Sun
ita','Kalpana','GN','Mukesh','Suresh','Rohil','Ramesh','Manoj','Rama
n','Kannan','Sundar','Vishal','Amit','Kamlesh','Jenny']
fooditems=['Dosa','Pavbhaji','VadaPav','AlooParatha','BreadButter','
BreadJam','BhelPuri','PaneerTikka','ShahiPaneer','Kadhi','Biryani','
FriedRice','Pizza','Burger','Sandwich']
people=students+teachers
courses=['DSA','DBS','MuP','DD','POE','POM','DISCO','OOP','PAVA','CP
','ES','TRW','MeOW','Thermo','M1','M2','M3','PnS','Bio','PPL','CompA
rch','DAA']

METADATA_FILENAME='metadata.txt'

#This takes the cross product of two lists and then selects a subset
of them essentially creating a relation
def createCrossData(list1,list2):
    cross=[]
    for l1 in list1:
        for l2 in list2:
            #Randomly choose whether or not to keep this tuple in
the final relation
            if random.randint(0,1)==1:
                cross.append((l1,l2))
    return cross

#Converts the contents of the crosstable (l1,l2) from tuple format
to String format
def createTripletStrings(crossTable,predName):
    tripletList=[]
    for l1,l2 in crossTable:
```

```
        tripletList.append(l1+'|'+predName+'|'+l2)
    return tripletList

#Takes a list of strings as input and outputs each on a new line in
the specified file
def printStringListToFile(stringList,filename):
    file = open(filename,'w')
    for string in stringList:
        file.write(string)
        file.write('\n')

#Checks if the given key is already present as a subject in the
MetaData file
def isSubjectPresent(key):
    fhandle=open(METADATA_FILENAME,'r')
    #Reads all lines of the metadata file
    lines=fhandle.readlines()

    for line in lines:
        line=line.rstrip()
        items=line.split('|')
        #Split the triplet into 3 and choose the first one(subject)
        subject=items[0]
        if subject==key:
            fhandle.close()
            return True
    fhandle.close()
    return False

#Prints the student metadata
def printStudentMetadata(crossTable1,crossTable2,crossTable3):
    f=open(METADATA_FILENAME,'a+')
    string='|memberOf|student'

    #Adds all students whose names appear in any relation to the
metadata file
    for t,s in crossTable1:
        #Make sure the file is closed before opening in another mode
        if f.closed==False:
            f.close()
        #If the name is not already present in the metadata file,add
it.
        if isSubjectPresent(s)==False:
            f=open(METADATA_FILENAME,'a+')
            f.write(s+string+'\n')

    for s,fo in crossTable2:
        if f.closed==False:
```

```
            f.close()
        if students.count(s)!=0 and isSubjectPresent(s)==False:
            f=open(METADATA_FILENAME,'a+')
            f.write(s+string+'\n')

    for s,co in crossTable3:
        if f.closed==False:
            f.close()
        if isSubjectPresent(s)==False:
            f=open(METADATA_FILENAME,'a+')
            f.write(s+string+'\n')
#Prints the teacher metadata
def printTeacherMetadata(crossTable1,crossTable2):
    f=open(METADATA_FILENAME,'a+')
    string='|memberOf|teacher'
    #Adds all teachers whose names appear in any relation to the
metadata file
    for t,s in crossTable1:
        #Make sure the file is closed before opening in another mode
        if f.closed==False:
            f.close()
        #If the name is not already present in the metadata file,add
it.
        if isSubjectPresent(t)==False:
            f=open(METADATA_FILENAME,'a+')
            f.write(t+string+'\n')

    for s,fo in crossTable2:
        if f.closed==False:
            f.close()
        if teachers.count(s)!=0 and isSubjectPresent(s)==False:
            f=open(METADATA_FILENAME,'a+')
            f.write(s+string+'\n')
#Prints the fooditems metadata
def printFoodMetadata(crossTable1):
    f=open(METADATA_FILENAME,'a+')
    string='|memberOf|foodItem'
    #Adds all fooditems whose names appear in any relation to the
metadata file
    for s,fo in crossTable1:
        #Make sure the file is closed before opening in another mode
        if f.closed==False:
            f.close()
        #If the name is not already present in the metadata file,add
it.
        if isSubjectPresent(fo)==False:
            f=open(METADATA_FILENAME,'a+')
            f.write(fo+string+'\n')
```

```
#Prints the courses metadata
def printCourseMetadata(crossTable1):
    f=open(METADATA_FILENAME,'a+')
    string='|memberOf|course'
    #Adds all courses whose names appear in any relation to the
metadata file
    for s,co in crossTable1:
        #Make sure the file is closed before opening in another mode
        if f.closed==False:
            f.close()
        #If the name is not already present in the metadata file,add
it.
        if isSubjectPresent(co)==False:
            f=open(METADATA_FILENAME,'a+')
            f.write(co+string+'\n')

#Create some dummy relations
teaches=createCrossData(teachers,students)
eats=createCrossData(people,fooditems)
registeredIn=createCrossData(students,courses)

#Generate triplets from the relations
tripTeaches=createTripletStrings(teaches,'teaches')
tripEats=createTripletStrings(eats,'likesToEat')
tripReg=createTripletStrings(registeredIn,'registeredIn')

#Take all triplet strings into a single list and output them to a
file
tripAll=tripTeaches+tripReg+tripEats
printStringListToFile(tripAll,filename='sampleData.txt')

#print Metadata to a file METADATA_FILENAME
printStudentMetadata(teaches,eats,registeredIn)
printTeacherMetadata(teaches,eats)
printFoodMetadata(eats)
printCourseMetadata(registeredIn)
```

Code (Shell script) to convert text file to GDF file

```
#!/bin/bash

#This function parses the input file into 7-comlumn format to
generate GDF file
#The function also generates the meta data file

function xtoGDF {
    filename=$1
    outname=''
    udi=''

    #code to extract the primary name of the file
    IFS='.'                             #Set the delimiter as '.'
    fileArr=()
    read -ra fileArr <<< "$filename"    #Get the name of the file
without the extension
    outname="${fileArr[0]}.gdf"         #Create name of the output
file. It will have same name as input file with .gdf extension

    IFS='|'                             #Set the delimiter as '|'

    count=0

    ##This loop reads the input file line by line and generates the
GDF tuples in 7-column format
    while IFS= read -r line
    do
        array=()
        uidArr=()

        ############ Code to create 7 column format ##############
        read -ra array <<< "$line"      #Read the line and split it
using delimiter '|'

        ###After splitting#####
        ##  array[0]    is the  subject
        ##  array[1]    is the  subject qualifier
        ##  array[2]    is the  predicate
        ##  array[3]    is the  predicate qualifier
        ##  array[4]    is the  object
        ##  array[5]    is the  object qualifier

        cum="${array[0]} ${array[1]} ${array[2]}"
        uid=$(echo "$cum" | md5sum)         #uid is the md5sum of
Subject' 'Predicate' 'Object
```

```
        ###The output of md5sum is hash value and some other
information, both separated by space.
        ###Hence we will delimit the uid by ' '
        read -ra uidArr <<< "$uid"
        IFS='|'

        uid=${uidArr[0]}


        #toWrite contains the tuple in 7-column format

toWrite="$uid|${array[0]}|${array[3]}|${array[1]}|${array[4]}|${arra
y[2]}|${array[5]}"

        if [[ $count -eq 0 ]]           #For the 1st iteration,
create the GDF file
        then
            echo "$toWrite" > "$outname"
        else                            #For other iterations,
append to the GDF file
            echo "$toWrite" >> "$outname"

        fi

        let count=$count+1
    done < "$1"


    ###### Below code is to generate the metadata.GDF file

    IFS='|'         #Set the delimiter to '|'
    let count=0
    while IFS= read -r line     #Read the metadata.txt file line by
line
    do
        read -ra array <<< "$line"          #Split the line by '|'

        ### array[0]    is the  Subject/Object/Predicate
        ### array[1]    is the  memberOf
        ### array[2]    is the  Node Type or Attribute Type
        cum="${array[0]} ${array[1]} ${array[2]}"
        uid=$(echo "$cum" | md5sum)             #Again generated UID
as the md5 sum of array[0]' 'array[1]' 'array[2]
        id=$(echo "${array[0]}" | md5sum)       #A unique ID is also
generated for Subject/Object/Predicate

        ###Again, since md5sum gives hash and some other info
delimited by ' ', we split by ' ' delimiter
```

```
        IFS=' '
        read -ra uidArr <<< "$uid"
        read -ra idArr <<< "$id"
        IFS='|'


        uid=${uidArr[0]}
        id=${idArr[0]}

        #toWrite contains the tuple (7-column format) to be written
in GDF
        #NOTE: qualifier for the subject/object/predicate in this
file will be a unique id already generated

        toWrite="$uid|${array[0]}|$id|${array[1]}||${array[2]}"

        if [[ $count -eq 0 ]]                   #For the 1st
iteration create the file
        then
            echo "$toWrite" > "metadata.gdf"
        else                                    #For other
iterations append to the file
            echo "$toWrite" >> "metadata.gdf"
        fi
        let count=$count+1
    done < "$2"

    #set the IFS back to default
    IFS=' '
}

xtoGDF $@           #Call the function xtoGDF with all the arguments
```

ReadMe file for the parser (Text to GDF convertor)

```
SUBJECT: Text to GDF Convertor
--------------------------------------------------------------------------------
RELATED FUNCTIONS:
                    xtoGDF()    :  Requires filename of the text file as
the only command line argument
--------------------------------------------------------------------------------


OUTPUT:
                    The command is a silent command. A new file with the
same name as the input file (but with .gdf extension) will be generated
containing the GDF format
--------------------------------------------------------------------------------
SYNOPSIS:

Section 1: TEXT FILE FORMAT

    The text file must be formtted by the following rules:

    a) Each line must have only one entry
    b) Multiple enteries (i.e. lines) must be seperated by a newline (\n)
character
    c) Each line should have the following format:


Subject_Name|Predicate_Name|Object_Name|Subject_Qualifier|Predicate_Qualifi
er|Object_Qualifier

        NOTE:
        c.1) All or any enteries can be omitted

        c.2) If an entry from the middle of the format is omitted, even
then the delimiters (i.e. |) must be kept
             Eg: Yash|likes|IceCream|Intelligent||Chocolate

            But if the entry omitted is not in the middle of the format,
then successive enteries can be omitted too
            Eg: Yash|likes|IceCream

        c.3) If a line doesn't contain any entry, GDF format will be :
UID||||||
--------------------------------------------------------------------------------
Section 2: THE GDF FORMAT

    The output file has the following format:

    a) Corresponding to each entry(i.e. line) of the text file, an entry is
written in .gdf file
    b) Multiple enteries are separated by newline character
    c) Each line has the following format:


UID|Subject_Name|Subject_Qualifier|Predicate_Name|Predicate_Qualifier|Objec
t_Name|Object_Qualifier

        NOTE:
        c.1) UID is generated as the md5sum hash of (SubjectName'
'Predicate_Name' 'ObjectName). Here ' ' indicates a space
```

```
        c.2) Based on the input, each entry can have any or all enteries
omitted. But delimiter '|' would still be present
--------------------------------------------------------------------------
```

about( ) : Function to create a file that contains the names of all the unique subjects

```
#!/bin/bash

#### Function to generate a file that will have name of all the
unique subjects

function about {
    filename=$1
    IFS='.'                                 #delimiter set to '.'


    fileArr=()
    read -ra fileArr <<< "$filename"        #Split the name of File
using '.' as the delimiter

    tmpname1=${fileArr[0]}_tmp1.gdf         #Create name of 1st
temporary file

    count=0

    #### This loop will copy the content of the GDF file as the
input into another temporary file ####

    while IFS= read -r line                 #Read the file line by
line
    do
        if [[ $count -eq 0 ]]               #For the 1st iteration,
create the file
        then
            echo "$line" > "$tmpname1"
        else
            echo "$line" >> "$tmpname1"     #For other iterations,
append to the file
        fi
        let count=$count+1
    done < "$1"


    tmpname2=${fileArr[0]}_tmp2.gdf         #Create name of the 2nd
temporary file
    sort -k 2,2 --field-separator='|' "$tmpname1" > "$tmpname2"
#Sort the data in 1st temporary file and store it in the 2nd
```

```
    IFS='|'                                 #Now, set delimiter as
'|'

    tmpname3=${fileArr[0]}_tmp3.gdf         #Create name of the 3rd
temporary file

    count=0

    ###Copy the subjects from the sorted data in 2nd temporary file
into the 3rd temporary file
    ###Since data in 2nd tmp file is delimited by '|', we can split
using this delimiter and store the result in array
    ###The 0th entity in array is the subject

    while IFS= read -r line
    do
        IFS='|'
        segment=()
        read -ra segment <<< "$line"

        if [[ $count -eq 0 ]]                   #For the 1st
iteration create the file
        then
            echo "${segment[1]}" > $tmpname3
        else                                    #For other
iterations, append to the file
            echo "${segment[1]}" >> $tmpname3
        fi
        let count=$count+1
    done < "$tmpname2"

    aboutFile=${fileArr[0]}_about.gdf       #Create the final output
file with name inputName_output.gdf
    uniq "$tmpname3" > "$aboutFile"         #Copy only the unique
data of 3rd tmp file into the output file
    rm $tmpname1 $tmpname2 $tmpname3        #Remove all the
temporary files
    IFS=' '                                 #Set the delimiter to
its default value
}

about $@            #Call the functions with all the arguments
```

separateBySub( ): Function to create a separate file for each subject having all information of that subject

```
#!/bin/bash
function separateBySub
{
    filename=$1
    outname=''
    rel=(NaN NaN NaN NaN NaN NaN NaN)
    IFS=$'\n'
    while read line                                    #Loop to go
through each relation in a gdf file
    do
        IFS='|' read -r -a rel<<<"$line"               #Reading the
line into array 'rel' separated by '|'
        local outfile=${rel[1]}.gdf
        flag=$(ls | grep $outfile)                     #Flag to
check for existence of the filename
        if [[ -z $flag ]]
        then
            touch $outfile                             #Creating
the file from subject's name if it doesn't exist
        fi
        echo $line >> $outfile                         #Appending
the relation
        IFS=$'\n'
    done < $filename
}

separateBySub $@                                       #calling the
function
```

REFERENCES

1. https://github.com/Sreyas-108/GDF : Link to the GIT repository having all our work

2. https://www.w3.org/TR/rdf-concepts/ : Official documentation for RDF

3. https://www.w3.org/TR/rdf-sparql-query/ : Official documentation for SPARQL (Query language for RDF)

4. https://en.wikipedia.org/wiki/SPARQL : For SPARQL

5. https://d3js.org/ : Official website of D3.js, the renderer we are using

6. https://wiki.dbpedia.org/ : Link to DBPedia – the site from where we will collect 3-column formatted data

7. Ian Robinson, Jim Webber & Emil Eifrem, *"Graph Databases"*, 2nd Edition

GLOSSARY

RDF: Stands for Resource Description Framework. It is a graph-based database system (NoSQL)

SPARQL: The query language for RDF

Meta-Data: The data about data

JSON: Stands for Java Script Object Notation. It is a lightweight format for storing and transporting data.

Renderer: It is a software to read a data file. In context of our project, renderer will read the JSON objects generated from the GDF file and generate Graph SVG.

SVG: Stands for Scalable Vector Graphics. It is an XML based vector image format that supports interactivity and animation.

D3.js: JavaScript library for manipulating documents based on data. This is the renderer we are using in our project.