SCALABLE CONVERSION OF TEXTUAL UNSTRUCTURED DATA TO NOSQL GRAPH REPRESENTATION USING BERKELEY DB KEY-VALUE STORE FOR EFFICIENT QUERYING

by

JASMINE MANOJ VARGHESE

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2017

Copyright © by Jasmine Manoj Varghese 2017

All Rights Reserved

Acknowledgements

I express sincere gratitude to my supervising professor, Dr. Ramez Elmasri who has been a motivating factor and a constant source of encouragement throughout my master's research. Without his guidance and excellent foresight this thesis would have only remained a great idea. I am sincerely thankful to Dr. Leonidas Fegaras and Prof. David Levine for their valuable suggestions and serving on my committee. I am also grateful to Dr. Sharma Chakravarthy for his role in the initial phase of my research.

I would like to thank the administrative staff, especially Ms. Pam McBride and Ms. Sherri Gotcher for their valuable support and services. Special thanks to Ms. Shalonda Towns in helping me through the final requirements of my thesis, and the department of Computer Science, UTA. I am grateful to Irie Bito for his support and for maintaining a well-administered research environment.

My heartfelt thanks to my family including my parents, my husband and kids for their patience and understanding, my brother and his family for their continuous support and inspiration.

I would like to express my appreciation to Soumyava Das, Neelabh Pant, Abhishek Santra, Jay D. Bodra, Aishwarya Ashok, Bhanu Jain and other friends in MAST and ITLAB. I also greatly appreciate my numerous other friends for their love and support.

November 13, 2017

Abstract

SCALABLE CONVERSION OF TEXTUAL UNSTRUCTURED DATA TO NOSQL GRAPH REPRESENTATION USING BERKELEY DB KEY-VALUE STORE FOR EFFICIENT QUERYING

Jasmine Manoj Varghese, MS

The University of Texas at Arlington, 2017

Supervising Professor: Ramez Elmasri

Graph database is a popular choice for representing data with relationships. It facilitates easy modifications to the relational information without the need for structural redefinition, as in case of relational databases. Exponentially growing graph sizes demand efficient querying, memory limitations notwithstanding. Use of indexes, to speed up query processing, is integral to databases. Existing works have used in-memory approaches that were limited by the main memory size. This thesis proposes a way to use graph representation, indexing technique and secondary memory to efficiently answer queries. Textual unstructured data is parsed to identify entities and assign unique identification. The entities and relationships are assembled into a graph representation in the form of key-value pairs. The key-value pairs are hashed into redundant Berkeley Database stores, clustered on relationships and entities. Berkeley DB key-value store uses primary memory in conjunction with secondary memory. Redundancy is affordable, since main memory size is not a limitation. Redundant key-value hash stores facilitate fast processing of many queries in multiple directions.

Contents

Table of Illustrations

List of Tables

Chapter 1

Introduction

This chapter discusses the growth of data and advantage of graph data structure. It explains the proposed work of scalable creation of query friendly graph format key-value store from textual unstructured data. Additionally, redundant storage of relationship information is used to answer many queries in multiple directions in constant time.

1.1 Graph Data Structure

Information-rich data sets are growing at an exponential rate, given the expanding avenues for data generation and collection. To store and retrieve information efficiently from these datasets, it is essential to find a representation that can capture the entities and their varying degree of relations. NoSQL graph data structure fills the void left by traditional relational representations, which was based on unchanging relationships. Batra and Tyagi [1] explain that graph databases can accept all types of data: structured, unstructured, and semi-structured - more easily than relational databases, which rely on a predefined schema.

Graph representation is adept at capturing varying degree of relations among the entities. Nodes represent the entities and edges represent the relationships. Graph can capture complex, non-systematic and hierarchical relationships. It provides an intuitive understanding of the relations. Increasing volumes of data, which are ideally suited for graph representation, reinforces this choice. The data structures used by NoSQL databases viz. key-value, wide column, document or graph are different from those used by default in relational databases, making some operations faster in NoSQL [2].

1.2 Information Retrieval

Knowledge rich data graphs have information that may be useful to businesses and individuals alike. It becomes imperative to efficiently and effectively query the information in these graph data stores. Query-processing techniques for graph databases is still in its nascent stages as compared to techniques available for relational databases. Existing graph querying approaches are memory based. Memory limitations become a bottleneck given the growing sizes of graphs. In addition, they convert graph representation to some form of non-graph representations for query processing. To take advantage of relational model's querying techniques, requires the graph to be converted to an equivalent relational representation. This involves overhead for the conversion and it is also sensitive to minor changes in relationships. Expensive join operations are employed to retrieve the relationship information. This narrows any advantages.

1.3 Proposed Work

Non-proliferation of approaches for querying graph databases in native graph representation led to the idea of using relational technique viz. hash indices for graph query processing. Indexing technique is an established method for efficient querying. Creating an indexed representation will reduce the time for processing queries. Clustering is a technique that groups data based on a similarity index. Clustering the nodes based on node-types and relationships in addition to indexing makes it easier to answer a query. This thesis aims to convert unstructured data to redundant query-friendly graph stores having key-value pairs using an external hash index that can be used to answer popular queries.

"Berkeley DB (BDB) is a software library intended to provide a high-performance embedded database for key-value data. Berkeley DB is written in C with API bindings for

C++, C#, Java, Perl, PHP, Python, Ruby, Smalltalk, Tcl, and many other programming languages.” [3]

Dataset used in this thesis is Internet Movie Database. The dataset consists of more than forty files. A handful of files, that contain the core information, have been used for this thesis. Source data file is read one line at a time. Entities are identified from the line using patterns. A unique identification is generated using combination of a letter for every entity (node-type) and a unique identification number for every entity value (node). This alphanumeric identification (ID) is unique across the dataset. The ID is stored in the entity’s ID hash index and reused for generation of graph information involving this entity. The (node,ID) pair is used to generate graph description consisting of “node1-node1ID-node2-node2ID-relationship”. A reversed combination “node2-node2ID-node1-node1ID-relationship” is also generated. The redundant storage helps to answer queries in multi direction.

This approach is not explored to the best of my knowledge.

Chapter 2

Data And Tools

This chapter lists the data and tools that are used. IMDB dataset contains information on titles and people associated with movies and TV series. Berkeley database API for Java is a library that creates and manages a high-performance embedded database.

2.1 IMDb

Domains that typically generate data suitable for graph representation has seen a sharp increase in recent years. This has resulted in several knowledge graphs. Knowledge graphs viz. DBLP [4] (a computer science bibliography, has nearly 6 million records of publications, authors, conference and journal articles), Freebase [5] (a collection of structured data harvested from various sources has 1.9 billion triples), Internet Movie Database (IMDB) (contains information of about 4.6 million movie titles including episodes and 8.2 million people associated with movies, TV-series etc.), to name a few. The Internet Movie Database (abbreviated IMDb) is an online database of information related to films, television programs and video games, including cast, production crew, fictional characters, biographies, plot summaries, trivia and reviews, operated by IMDb.com. As of Nov 2017 it contains nearly 4.6 million titles including episodes and 8.2 million related people. Registered users are invited to submit new entries and edits to existing ones [6]. Until October 2017 the dataset was available for free download on IMDB.com for private or research purposes. The information is spread over multiple files. Each file contains data related to a single aspect of information. The files are maintained by multiple persons. Data in its current form is not amenable to querying. Sufficient effort was made to enforce a

uniform format with multiple delimiters. The varying nature of information in each file has introduced minor deviations. All the information is contained in over fifty files. In this thesis, nine files were used that contained the crux of the data. Data pertains to movies and episode titles, year in which they were produced, actors, actresses, producers, directors, country and locations in which the scenes were shot, language spoken and their genres. A node-type is equivalent to an entity. The entities present in the files are movie, episode, TV series, year, actor, actress, director, producer, language, city, state, country and genre. Every occurrence of a node-type translates as a node. For example, “Clooney, George” is a node of node-type actor and “Superman” is a node of node-type movie.

2.2 Berkeley DB (BDB)

Berkeley DB is an open source library that can create and manage an embedded database. It supports scalable, high-performance, transaction-protected data management services within the application. Berkeley DB provides simple function-call API for database management.

BDB supports key-value datatype. It offers proven reliability and availability. Berkeley DB is designed to provide heavy duty database services to application developers, without requiring database expertise. At its core is a C-library style *toolkit* that provides a broad base of functionality to application developers [7].

Further, Berkeley DB is designed to interact correctly with the native system's toolset, a feature no other database package offers. For example, on UNIX systems Berkeley DB supports hot backups (database backups while the database is in use), using standard UNIX system utilities, for example, dump, tar, cpio, pax or even cp. On other systems, which do not support filesystems with read isolation, Berkeley DB provides a tool for safely copying files.

Availability of various scripting interfaces for Berkeley DB encourages faster application development.

BDB allows applications to specify a suitable storage structure. BDB supports hash tables, Btrees, record-number based and queue storage structures. Hash table storage structure has predictable search and update times for random-access records. This structure is ideal for exact matches rather than range matches.

Berkeley DB generates database files on disk that can be archived for later use. This feature helps in amortizing the cost of creating hash stores once while processing several queries later.

2.3 Java

Java is a platform independent, strongly-typed object-oriented language. Java being a compiled language, makes for faster executions making it better suited to handle scalability and portability. It also has rich collection of data structures and APIs for data manipulation.

Chapter 3

Generation Of Master Files

This chapter explains the process of generating two categories of Berkeley-DB hash files. The first category is an ID hash file for each node-type that stores (key,value) pairs of (node-value, ID). The second category is a graph hash file that holds graph information as (key,value) pairs of (node-value, relation information with another node-value).

3.1 Identification Generation

A unique alphanumeric identification is generated and assigned to every node. Each entity (node-type) present in the dataset is assigned a pre-defined unique letter for identification. Upon retrieving a node from a line in the source file, it is hashed into its respective Berkeley-DB hash file for ID. If the node (key) is present, the ID (value) is retrieved and reused to generate the graph information. If the node (key) is absent, then it is assigned the next sequence number for that entity (node-type). The (key,value) pair of (node, ID) is stored into the Berkeley-DB hash file for ID. Eg. movie entity (node-type) is assigned letter ‘m’. A movie ‘Mission Impossible” may be assigned ‘m100453’ as ID. The (key,value) pair of (‘Mission Impossible’,’m100453’) is inserted in “moviesId.db”. No other movie will be assigned this number. The alphanumeric combination makes it unique across the dataset. This ID is used to generate graph information and reused for all subsequent occurrences of the movie across the dataset.

3.2 File Processing

A line from the source file is processed to extract the node values based on pre-defined patterns. The node value is hashed into the respective node-type ID hash table to retrieve Id. If the value is present in the hash table, then the Id is retrieved and used to form the graph information. Otherwise, a new Id is generated and the pair (node-value, node-Id) is stored in the ID hash table. The time complexity is O(1) to check and retrieve if the item is already present in the hash table. Hashing allows for efficient retrieval of Id considering the scalability factor. One Berkeley-DB hash table is used to store ID for one node-type. A section of file “movies.list” is shown in *Figure 3*-1. String enclosed in quotes indicate TV series name. The first sequence of year enclosed in parenthesis indicate the year in which the TV series was produced. String enclosed in braces indicate the episode name. The second year enclosed in parenthesis indicate the year in which the episode was produced. Using these values graph information is generated having key as “<node1>” and value as “<node1Id> <node2IValue> <node2Id> <relation>”. Reverse graph information is also generated, to enable query processing in both ways. Reverse format consists of key “<node2>” and value “<node2Id> <node1> <node1Id>.

```
1  "!Next?" (1994)                    1994-1995
2  "#1 Single" (2006)                 2006-????
3  "#1 Single" (2006) {Cats and Dogs (#1.4)}       2006
4  "#1 Single" (2006) {Finishing a Chapter (#1.5)}     2006
5  "#1 Single" (2006) {Is the Grass Greener? (#1.1)}   2006
6  "#1 Single" (2006) {Stay (#1.8)}            2006
7  "#1 Single" (2006) {The Rules of Dating (#1.3)}     2006
8  "#1 Single" (2006) {Timing Is Everything (#1.7)}    2006
9  "#1 Single" (2006) {Window Shopping (#1.2)}     2006
10 "#1 Single" (2006) {Wingman (#1.6)}         2006
11 "#15SecondScare" (2015)                 2015-????
```

Figure 3-1 Contents of movies.list

3.2.1 Movies.list

"movies.list" contains names of movies, or tv series and episode name, and the year in which they were produced. "moviesId.db" is a BDB hash table that stores the (movie name, Id) pair of the "movies" node type. Character "m" is prefixed in the identification number. The Id is unique for each entry among all the type nodes. It is retrieved and reused for the same node value. Contents of "moviesId.db" is as shown in table 3-1. "moviesId.db" has 1120136 records after processing all the source files. Graph information for movies and years is generated and recorded in the "moviesYears.db" hash file with the format "<moviename>" as the key and "<movieId> <year> <yearId> " as the value. A reverse entry is made in "yearMovie.db" with key "<year>" and value "<yearid> <movieName> <movieId>". It may have multiple entries for a key. Storing information redundantly allows queries to be answered in either ways. "yearId.db" is the hash table for "year" type node. It stores the (year, Id) pair. A prefix of character "y" is used to generate unique identification. It records 160 years. "tvseriesId.db" is a BDB hash table for "tvseries" node type. It stores the (tv-series name, Id) and records 142964 entries. Graph information of tvseries with year is recorded in the hash file "tvseriesYear.db" with the format "<tvseriesname>" as the key and <tvseriesId> <year> <yearId>" as the value. A reverse entry key-value pair "<year>"-"<yearId> <tvseriesName> <tvseriesId>" is recorded in "yearTvseries.db". Episode-name node type is stored in "episodesId.db" BDB hash table containing (episode name, Id) pair. In all, 1507147 episodes were recorded in the table. The graph record for episodes and year has "<episodeName>" for the key and "<episodeId> <yearId> <year>" for the value and is stored in "episodeYear.db". A reverse key-value entry "<year>" – "<yearId> <episodeName> <episodeId>" is made in "yearEpisode.db". Tvseries and episode information is entered in "tvseriesEpisode.db" with key "<tvseriesName>" and value "<tvseriesId> <tvseriesYear> <episodeName>

<episodeId>". Inversely, key "<episodeName>" and value "<episodeId> <tvseriesName> <tvseriesId> <tvseriesYear>" is recorded in "episodeTvseries.db".

Table 3-1 Contents of "movieID.db"

movieName	**movieId**
Zanzibar pittoresque	m1109717Z
ai na he pan qing cao qing	m1109252
Your Name	m1106820
Yami Douga 6	m1100332
Why Fight Death	m1086083
Ván bài lat ngua: Tap 1 - Dua con nuoi vi giám muc	M1066549
Virupakshuni Vichitra Guha	m1061344
Valladesam	m1051402
Une de la cavalerie	m1040930
…….	

3.2.2 Actors.list

"actors.list" contains the name of actors and the movies, or TV series and episodes, that they acted in. "actors" is a node-type and character "r" is used for identification prefix. Key-value pair (actorname, actorId) is stored in "actorId.db" BDB hash table. It generates 2608412 entries. Movie and actor graph information is recorded in "movieActor.db" with key "<movieName>" and value "<movieId> <movieYear> <actorName> <actorId > <characterRole>". Reverse entry for actor-movie is recorded in "actorMovie.db" with key "<actorName>" and value "<actorId> <movieName> <movieId> <movieYear> <characterRole>". Episode and actor graph information is recorded in "episodeActor.db" with key "<episodeName>" and value "<episodeId> <actorName>

<actorId> <characterRole>". Reverse key-value pair is entered in "actorEpisode.db" with key "<actorName>" and value "<actorId> <episodeName> <episodeId> <characterRole>". Actors and TV series graph information is recorded in "tvseriesActor.db" with key "<tvseriesName>" and value "<tvseriesId> <tvseriesYear> <actorName> <actorId>". A reverse entry is recorded in "actorTvseries.db" having key "<actorName>" and value "<actorId> <tvseriesName> <tvseriesId> <tvseriesYear>".

3.2.3 Producers.list

"producers.list" contains the name of producers of movies or TV series and episodes that they produced. "producer" is a node-type with character "p" prefixed for identification. (producer name, Id) pair is stored in "producerId.db" BDB hash table and records 847592 producers. The file also contains sub categories of producer viz. executive-producer, which is accommodated as an edge information. Movies and producers graph information is hashed in "movieProducer.db" with key "<movieName>" and value "<movieId> <movieYear> <producerName> <producer Id> <producerSubcategory>". Episodes and producer information is translated into graph data having key "<episodeName>" and value "<episodeId> <producerName> <producerId> <producerSubcategory>" and hashed in "episodeProducer.db". Reverse key-value pair "<producerName>" – "<producerID> <episodeName> <episodeID> <producerSubcategory>" is hashed in "producerEpisode.db". TV series and producer graph information is hashed in "tvseriesProducer.db" with key "<tvseriesName>" and value "<tvseriesId> <tvseriesYear> <producerName> <producerId> <producerSubcategory>". Reverse key-value pair "<producerName>" – "<producerID> <tvseriesName> <tvseriesID> <tvseriesYear>" is stored in "producerTvseries.db".

3.2.4 Actress.list

“actresses.list” contains the name of actresses and the movies or, tv series and episode, that they acted in. “actresses” is a node-type and uses character “s” for alphanumeric identification. The key-value pair (actress name, actress Id) is stored in “actressId.db” BDB hash table. 1433091 entries are recorded. Movie and actress graph information is hashed in “movieActress.db” with key “<movieName>” and value ‘‘<movieId> <movieYear> <actress name> <actress Id> <characterRole>”. Reverse key-value pair is hashed in “actressMovie.db” with key “<actressName>” and value “<actressId> <movieName> <movieId> <movieYear>”. Episode and actress graph information is hashed in “episodeActress.db” with key “<episodeName>” and value “<episodeId> <actressName> <actressId>”. Reverse key-value pair is hashed in “actressEpisode.db” with key “<actressName>” and value “<actressId> <episodeName> <episodeId>”. Actresses and tv-series graph information is hashed in “tvseriesActresses.db” with key “<tvseriesName>” and value “<tvseriesId> <tvseriesYear> <actressName> <actressId>”. Reverse key “<actressName>” and value “<actressId> <tvseriesName> <tvseriesId> <tvseriesYear>” information is hashed in “actressTvseries.db”

3.2.5 Directors.list

“directors.list” lists the director of movies or, tvseries and episodes. “director” is a node-type using character “d” for unique identification. “directorId.db” is the BDB hash table that stores the (director name, director Id) pair. 481966 names of directors are recorded. Graph information of movies and directors is stored in “movieDirector.db” with key “<movieName>” and value “<movie Id> <movie name> <movieYear> <director Id> <director name>”. Reverse graph information is stored in “directorMovie.db” with key “<directorName>” and value “<directorID> <movieName> <movieID> <movieYear>”.

Episode and director graph information is stored in “episodeDirector.db” having key as “<episode name>” and value as “<episodeId> <director name> <director Id>”. Reverse key “<directorName>” and value “<directorId> <episodeName> <episodeId>” is hashed in “directorEpisode.db”. TV series and director graph is stored in “tvseriesDirector.db” with key “<tvseriesName>” and value “<tvseriesId> <tvseriesYear> <directorName> <director Id>”. Reverse information with key “<directorName>” and value “<directorId> <tvseriesName> <tvseriesId> <tvseriesYear>” is hashed in “directorTvseries.db”.

3.2.6 Countries.list

“countries.list” lists the countries where scenes were shot for each of the movie, or TV series and episodes. “country” is a node-type prefixing “c” for unique identification. “countryId.db” is a BDB hash table that stores (countryName, Id) pair. After processing, 220 countries were recorded. Graph data for movies and countries is hashed in “movieCountry.db” with key “<movieName>” and value “<movieId> <movieYear> <countryName> <countryId>”. Reverse key “<countryName>” and value “<countryId> <movieName> <movieId> <movieYear>” is hashed in “countriesMovie.db”. Episode and country graph information is stored in “episodeCountry.db” with key “<episodeName>” and value “<episodeId> <countryName> <countryId>”. Reverse key “<countryName>” and value “<countryId> <episodeName> <episodeId>” is hashed in “countryEpisode.db”. TV series and countries graph information is stored in “tvseriesCountry.db” with key “<tvseriesName>” and value “<tvseriesId> <tvseriesYear> <countryName> <countryId>”. Reverse key “<countryName>” and value “<countryId> <tvseriesName> <tvseriesId> <tvseriesYear>” is hashed in “countryTvseries.db”.

3.2.7 Languages.list

"languages.list" contains the different languages spoken in the movies, tv series and episodes. "language" is a type of node and character "l" is used as identification prefix. The key-value pair (language, Id) is stored in "languageId.db" BDB hash table. 358 languages are identified from the file. Movies and languages graph information is hashed in "moviesLanguage.db" having key "<movie name>" and value "<movieId> <movieYear> <language> <language Id>" Reverse key "<language>" and value "<languageId> <movieName> <movieId> <movieYear>" is hashed in "languageMovie.db". Episodes and languages graph information is hashed in "episodeLanguage.db" with key "<episodeName>" and value "<episode Id> <language> <language Id>". Reverse key "<language>" and value "<languageId> <episodeName> <episodeId>" is hashed in "languageEpisode.db". TV series and languages graph information is hashed in "tvseriesLanguage.db" having key "<tvseriesName>" and value "<tvseriesId> <tvseriesYear> <language> <language Id>". Reverse key "<language>" and value "<<languageId> <tvseriesName> <tvseriesId> <tvseriesYear>" is hashed in "languageTvseries.db".

3.2.8 Genres.list

"genres.list" contains the genres to which the movies, or tvseries and episodes belong. (genre, Id) is stored in "genreId.db" BDB hash table, for "genre" type node. Character "g" is used to generate the alphanumeric identification. After processing, 35 genres were identified. Movies and genres graph information is stored in "movieGenre.db" having key "<movieName>" and value "<movieId> <movieYear> <genre> <genreId>". Reverse key "<genre>" and value "<genreId> <movieName> <movieId> <movieYear>" is hashed in "genreMovie.db". Episodes and genres graph information is stored in

“episodeGenre.db” having key “<episodeName>” and value “<episodeId> <genre> <genreId>”. Reverse information is hashed in “genreEpisode.db” with key “<genre>” and value “<genreId> <episodeName> <episodeId>”. TV series and genres graph information is stored in “tvseriesGenres.db” having key “<tvseriesName>” and value “<tvseriesId> <tvseriesYear> <genre> <genreId>”. Reverse key “<genre>” and value “<genreId> <tvseriesName> <tvseriesId> <tvseriesYear>” is hashed in “genreTvseries.db”.

3.2.9 Locations.list

“locations.list” contains information on the locations for each of the movies, TV series and episodes. The location information consists of at most four sections viz. country, state, city and street address. Additional information about the location may be present in parenthesis.

```
"#LawstinWoods" (2013) {The Happening (#1.3)}       Erin, New York, USA
"#LawstinWoods" (2013) {The Happening (#1.3)}       Corning, New York, USA
"#LawstinWoods" (2013) {The Happening (#1.3)}       Poultney, Vermont, USA
"#LawstinWoods" (2013) {The Neverending Day} {{SUSPENDED}}  Corning, New York, USA
"#LawstinWoods" (2013) {The Neverending Day} {{SUSPENDED}}  Erin, New York, USA
"#LdnOnt" (2012)                    London, Ontario, Canada (studio)
"#LoveMyRoomie" (2016)                  New York, USA   (on location)
"#Millennials" (2015)                   Los Angeles, California, USA    (Steve and Debr
"#Mittelfingerspitzengefühl" (2017)         Germany (studio)
"#Mittelfingerspitzengefühl" (2017)         The Netherlands (location)
"#MommasGotBars" (2015)                 Brooklyn, New York, USA (Young Son's apartment)
"#Murder" (2017)                    Knoxville, Tennessee, USA
"#MyCurrentSituation: Atlanta" (2016)           Atlanta Georgia, USA
"#MyCurrentSituation: Atlanta" (2016)           Memphis, Tennessee, USA
"#NerdNoise" (2012)                 Los Angeles, California, USA    (on location)
"#Nightstrife" (2014)                   Chicago, Illinois, USA
"#OscarTheOuch" (2013)                  Hollywood, California, USA
```

Figure 3-2 Contents of locations.list

cityId.db stores the key-value (city, Id). Letter ‘a’ is prefixed with number to generate the unique alphanumeric ID. “stateId.db” stores the key-value (state, Id). Letter ‘b’ is prefixed with number to generate the unique alphanumeric ID for “state” node.

countryId.db hash index is reused from the processing of “countries.list”. It stores the key-value (country, Id) and as mentioned in section 3.2.6 alphabet ‘c’ is used to generate the alphanumeric ID.

Location and movie information is stored in three graph combinations each with key as city, state and country. The value consists of all information sans the key. For lines containing movie title, graph information for city and movie is generated with key as “<cityName>” and value as “<cityId> <movieName> <movieId> <movieYear> <stateName> <stateId> <countryName> <countryId> <additional information>” and stored in “locCityMovie.db”. State and movie graph information is stored in “locStateMovie.db” with key “<stateName>” and value “<stateId> <movieName> <movieId> <movieYear> <cityName> <cityId> <countryName> <countryId> <additional Information>”. Country and movie graph information is stored in “locCountryMovie.db” with key “<countryName>” and value “<countryId> <movieName> <movieId> <movieYear> <stateName> <stateId> <cityName> <cityId> <additional information>”. This file is similar to “countryMovie.db”, which has countryName as key but it does not have the state and city portion of the information.

Reversed graph information movie-location is stored in “movieLocation.db” with key as “<movieName>” and value as “<movieId> <movieYear> <cityName> <cityId> <stateName> <stateId> <countryName> <countryId> <additional information>”.

Similarly, location and episode information is stored in three graph combinations, city-episode, state-episode and country-episode. City and episode graph information with key “<cityName>” and value “<cityId> <episodeName> <episodeId> <stateName> <stateId> <countryName> <countryId> <additional information>” is stored in “locCityEpisode.db”. State and episode graph information is stored in “locStateEpisode.db”

with key “<stateName>” and value “<stateId> <episodeName> <episodeId> <cityName> <cityId> <countryName> <countryId> <additional information>”. Country and episode graph information is stored in “locCountryEpisode.db”. Again this is similar to the “countryEpisode.db”, generated after processing “countries.list”, but without state and city information. Reversed graph information for episode and location is stored in “episodeLocation.db” with key “<episodeName>” and value “<episodeId> <cityName> <cityId> <stateName> <stateId> <countryName> <countryId> <additional Information>”.

Also, location and tvseries information is stored in three graph combinations, city-tvseries, state-tvseries and country-tvseries. City and tvseries graph information with key “<cityName>” and value “<cityId> <tvseriesName> <tvseriesId> <tvseriesYear> <stateName> <stateId> <countryName> <countryId> <additional information>” is stored in “locCityTvseries.db”. Graph information for state and tvseries is stored in “locStateTvseries.db” with key “<stateName>” and value “<stateId> <tvseriesName> <tvseriesId> <tvseriesYear> <cityName> <cityId> <countryName> <countryId> <additional information>”. Country and tvseries graph information is stored in “locCountryTvseries.db” with key “<countryName> and value “<countryId> <tvseriesName> <tvseriesId> <tvseriesYear> <cityName> <cityID> <stateName> <stateID> <countryName> <countryID> <additional Information>”. Again this is similar to “countryTvseries.db”, generated after processing “countries.list” but without state and city information. Reversed graph information for tvseries and location is stored in “tvseriesLocation.db” with key “<tvseriesName>” and value “<tvseriesId> <tvseriesYear> <cityName> <cityId> <stateName> <stateId> <countryName> <countryId> <additional Information>”.

3.3 Summary Of Files Processed And Generated

Table 3-2 Summary of files

Source File	Files generated	Contents in key-value pairs	unique Char	Number of records
movies.list	movieId.db	(Movie name, movie Id)	m	1120210
	yearId.db	(Year, year Id)	Y	160
	tvseriesId.db	(tvseries name, tvseries Id)	t	142,964
	episodeId.db	(episode name, episode Id)	e	1,507,147
	movieYear.db	Moviename, movieId-year-yearId		1,368,317
	yearMovie.db	Year, yearId-moviename-movieId		1,368,317
	tvseriesYear.db	TvseriesName, tvseriesId -year-yearId		151,895
	yearTvseries.db	Year, yearid - tvseriesName - tvseriesid		151,895
	episodeYear.db	EpisodeName, episodeId-year-yearId		2,851,406
	yearEpisode.db	Year, yearId-Episodename-EpisodeId		2,851,406
	tvseriesEpisodes.db	Tvseriesname, tvseriesId-tvseriesYear-episodename-episodeId		2,851,406
	episodeTvseries.db	EpisodeName, episodeId-tvseriesName-tvseriesid-tvseriesYear		2,851,406

Table 3-2 – Continued

countries.list	countryId.db	(countryName, country Id)	c	220
	movieCountry.db	Moviename, movieId-movieYear-countryname-countryId		1320482
	tvseriesCountry.db	TvseriesName, tvseriesId-tvseriesYear-countryname-countryId		150252
	episodeCountry.db	EpisodeName, episodeId-countryName-countryId		490091
	countryMovie.db	Country, countryId-movieName-movieId-movieYear		1320482
	countryTvseries.db	Country, countryId-tvseriesName-tvseriesId-tvseriesYear		150252
	countryEpisode.db	Country, countryId-episodeName-episodeId		490091
languages.list	languageId.db	(language, language Id)	l	358
	movieLanguage.db	Moviename, movieId-movieYear -language-languageid		1300458
	tvseriesLanguage.db	Tvseriesname, tvseriesId-tvseriesYear -language-languageid		151690
	episodeLanguage.db	Episodename, episodeId -language-languageid		596312
	languageMovie.db	Language, languageId-movieName-movieId-movieYear		1300458
	languageEpisode.db	Language, languageId-episodeName-episodeId		596312

Table 3-2 – Continued

	languageTvseries.db	Language, languageId-tvseriesName-tvseriesId-tvseriesYear		151690
genres.list	genreId.db	(genre, genre Id)	g	35
	moviesGenres.db	Moviename, movieId-movieYear-genre- genreId		2327682
	tvseriesGenre.db	Tvseriesname, tvseriesId-tvseriesYear-genre-genreId		187452
	episodesGenre.db	Episodename, episodeId-genre-genreId		21218
	genreMovie.db	Genre, genreId-movieName-movieId-movieYear		2327682
	genreEpisode.db	Genre, genreId-episodeName-episodeId		21218
	genreTvseries.db	Genre, genreId-tvseriesName-tvseriesId-tvseriesYear		187452
actors.list	actorId.db	(actor name, actor Id)	r	2608412
	movieActor.db	Moviename, movieId-movieYear-actorname-actorId-character		7017252
	tvseriesActor.db	Tvseriesname, tvseriesId-tvseriesYear-actorname-actorId-character		12271979
	episodeActor.db	Episodename, episodeId-actorname-actorId-character		11846029

Table 3-2 – Continued

	actorMovie.db	actorName, actorId-movieName-movieId-movieYear-character		7017252
	actorEpisodedb	actorName, actorId-episodeName-EpisodeId-character		11846029
	actorTvseries.db	actorName, actorId-tvseriesName-tvseriesId-tvseriesYear- character		12271979
actresses.list	actressId.db	(actressname, actress Id)	s	1433091
	movieActress.db	Moviename, movieId - movieYear-actressname-actressId-character		3697518
	tvseriesActress.db	Tvseriesname, tvseriesId-tvseriesYear-actressname-actressId-character		8131251
	episodeActress.db	Episodename, episodeId-actressname-actressId-character		7855689
	actressMovie.db	actressName, actressId-movieName-movieId-movieYear-character		3697518
	actressEpisode.db	actressName, actressId-episodeName-episodeId-character		7855689
	actressTvseries.db	actressName, actressId-tvseriesName-tvseriesId-tvseriesYear-character		8131251
producers.list	producers.db	(producer name, producer Id)	p	847592

Table 3-2 – Continued

	movieProducer.db	Moviename, movieId-movieYear - producername - producerId – producertype		2080036
	tvserieProducer.db	Tvseriesname, tvseriesId-tvseriesYear-producername-producerId-producertype		5262419
	episodeProducer.db	episodename, episodeId-producername-producerId-producertype		4865565
	producerMovie.db	producerName, producerId-movieName-movieId-movieYear-producertype		2080036
	producerEpisode.db	producerName, producerId-episodeName-episodeId-producertype		4865565
	producerTvseries.db	producerName, producerId-tvseriesName-tvseriesId-tvseriesYear-producertype		5262419
directors.list	directorId.db	(director name, director Id)	d	481966
	movieDirector.db	Moviename, movieId-movieYear-directorname-directorId		1288140
	tvseriesDirector.db	Tvseriesname, tvseriesId -tvseriesYear-directorname-directorId		1580966

Table 3-2 – Continued

	episodeDirector.db	Episodename, episodeId - directorname-directorId		1478035
	directorMovie.db	directorName, directorId-movieName-movieId-movieYear		1288140
	directorEpisode.db	directorName, directorId-episodeName-episodeId		1478035
	directorTvseries.db	directorName, directorId-tvseriesName-tvseriesId-tvseriesYear		1580966
locations.list	locCity.db	(cityName, cityId)	a	54126
	locState.db	(stateName,stateId)	b	15173
	locCityMovie.db	cityName, cityId-movieName-movieId-movieYear-stateName-stateId-countryName-countryId-additionalInformation		668526
	locStateMovie.db	stateName, stateId-movieName-movieId-movieYear-cityName-cityId-countryName-countryId-additionalInformation		1161828
	locCountryMovie.db	countryName, countryId-movieName-movieId-movieYear-cityName-cityId-stateName-stateId-additionalInformation		1161828

Table 3-2 – Continued

	movieLocation.db	movieName, movieId-movieYear- cityName-cityId-stateName-stateId-countryName-countryId-additionalInformation		668526
	locCityEpisode.db	cityName, cityId-episodeName-episodeId-stateName-stateId-countryName-countryId-additionalInformation		416595
	locStateEpisode.db	stateName, stateId-episodeName-episodeId-cityName-cityId-countryName-countryId-additionalInformation		416595
	locCountryEpisode.db	stateName, stateId-episodeName-episodeId-cityName-cityId-countryName-countryId-additionalInformation		416595
	episodeLocation.db	episodeName, episodeId-cityName-cityId-stateName-stateId-countryName-countryId-additionalInformation		416595
	locCityTvseries.db	cityName, cityId-tvseriesName-tvseriesId-tvseriesYear-stateName-stateId-countryName-countryId-additionalInformation		493302

Table 3-2 – Continued

	locStateTvseries.db	stateName, stateId-tvseriesName-tvseriesId-tvseriesYear-cityName-cityId-countryName-countryId-additionalInformation		493302
	locCountryTvseries.db	countryName, countryId-tvseriesName- tvseriesId-tvseriesYear-cityName-cityId-stateName-stateId-additionalInformation		493302
	tvseriesLocation.db	tvseriesName, tvseriesId-tvseriesYear-cityName-cityId-stateName-stateId-countryName- countryId-additionalInformation		493302

3.4 Optimization

Large source files affected the processing time for conversion. The reason, probably, being that hash indexing technique involves expansion of bucket-sizes requiring rewrites. Couple of optimization techniques were introduced to speed up the process.

One technique was to process only one pair of entity combination in one pass of the source file instead of processing all the pairs of entity combination in one pass. This reduced the number of active hash indexes, thus improving speed. Eg. Processing directors.list (130Mb) for all node-types in one run on machine with 4GB RAM took 68 minutes. Whereas processing only movie-director and director-movie graph information took 9 minutes. Processing Episode-director and director-episode took 13 minutes.

Processing Tvseries-director and director-tvseries took 5 minutes. Processing all node-types individually in three different runs took only 27 minutes compared to the 68 minutes for processing all simultaneously in one run. Reducing the number of active Berkley DB hash files in an iteration reduced overall processing time.

For graph hash index files greater than 1GB, sharding (splitting) alphabetically reduced the size of the active hash index files thus improving speed. One entity pair combination required two passes to process. Eg. Processing producers.list (488Mb) to generate “producerEpisode.db” and “episodeProducer.db” on machine with 4GB RAM without sharding did not complete in four hours. Whereas processing all producer names starting with alphabets upto ‘J’, to generate “episodeProducer1.db” and “producerEpisode1.db”, in first run took 27 mins. And processing all producer names starting with alphabets after ‘J’, to generate “episodeProducer2.db” and “producerEpisode2.db” in second run took 36 minutes. The entire process taking just over an hour. Sharding on alphabets reduced the size of the active Berkley DB hash file to half in an iteration thus reducing processing time.

Sharding is fruitful when the hash file is proportionally larger than the available memory. On machine with 16GB RAM, sharding had no difference on processing time for Berkeley DB file upto 1.5 GB. Also, Retrieving data from smaller Berkeley DB hash is faster than from bigger files.

Eg. Retrieving first 50 rows on 4GB RAM machine from a 1.5 GB Berkeley DB hash file takes 736282 ms where from 800MB Berkeley DB hash file takes 370971 ms. On the downside sharding results in higher number of smaller files.

Chapter 4

Generation of Look-up data for person entity

This chapter discusses the process of generating look-up data for person entity from person related entities viz. actor, actress, producer and director.

Until now we have identified person related entities viz., “actor”, “actress”, “producer” and “director”. We can think of a “person” entity to associate all the person related entities. In IMDb data, names of actor, actress, producer and director contribute to form the person entity. It becomes easy to find the different roles of a person without having to go through each of the actor, actress, producer and director tables. Index like information is maintained to facilitate quicker response. Information related to the “person” entity is catalogued in one file. A person may have one or more of the roles viz. director, producer, actor and actress. In the person file, a person will have a “person” type Id as well as other type Ids corresponding to his roles. As an example, a person who is only an actor will have (key, value) as (<person name>, <personId><ActorId>). A person who is a producer and director will have (key,value) pair as (<person name>, <personId><DirectorId><ProducerId>). Person Id is prefixed with character “n” for unique identification across all the node types. “person” information is collected simultaneously during the processing of the actor, actress, director and producer files., When processing “actor” type node, the actor name is hashed and then retrieved or stored in “actors.db”. Parallelly, it is also hashed and stored in “person.db”. Since every “actorname” is present in the source file only once, we do not check for its existence before storing. The process is repeated for “actress” type node, where the actress name is hashed and stored in “person.db”. When processing director’s name from “directors.list”, the director’s name is

hashed in “person.db” and checked for existence. If found, then the (key,value) pair would be of the form (person name, <personId><ActorId/ActressId>). The value is appended with “<directorId>” and stored back. The updated (key,value) pair would be of the form (<person name>, producerId><ActorId/ActressId><DirectorId>). If the “directorname” is not found in “person.db” then a new person Id is generated and the (key,value) pair (<personname>,<personId><directorId>) is hashed and stored into “person.db”. Producer type node is processed similarly. On completion a person who is a producer, director and an actor will have a (key,value) pair of the format (<person name>, <person Id> <ActorId/ActressId> <directorId> <producerId>). A person who is only a producer will have a (key,value) pair of the form (<person name>, <personId> <producerId>).

Sample rows of “person.db” BDB hash table

Table 4-1 Contents of person.db

Key	**Value**			
Person name	Person Id	Actor /Actress Id	Director Id	Producer Id
Clooney, George	(n440044)	(r440044)	(d81116)	(p141368)
Cicoria, Tony	(n427048)	(r427048)		
Christy (XI)	(n2839921)	(s231509)		
Wood-Hill, Matthew	(n4334131)		(d469596)	

Chapter 5

Queries

This chapter describes the queries that can be answered using the hashed key-value stores.

Information retrieval is effective only when an efficient query processing technique is present. Hash indexing guarantees constant time in retrieving records. This is the prime reason for converting the unstructured data to redundant hashed key-value stores. Following is a sample of the type of queries that can be answered using the hashed key-value graph stores. Operations similar to join can also be performed with iterative retrievals.

5.1 Query Ttype 1

Q. Display the countries in which Poverty, Inc. movie scenes were shot

Poverty, Inc.: USA
Poverty, Inc.: UK
Poverty, Inc.: Thailand
Poverty, Inc.: Swaziland
Poverty, Inc.: Korea
Poverty, Inc.: Africa
Poverty, Inc.: Rwanda
Poverty, Inc.: Russia
Poverty, Inc.: Peru
Poverty, Inc.: Kenya
Poverty, Inc.: Italy
Poverty, Inc.: Ireland
Poverty, Inc.: India
Poverty, Inc.: Haiti
Poverty, Inc.: Georgia
Poverty, Inc.: Ethiopia
Poverty, Inc.: Republic
Poverty, Inc.: Cambodia
Poverty, Inc.: Bangladesh
Poverty, Inc.: Argentina

The query is answered by hashing for “Poverty, Inc” into “movieCountry.db”. The database is set to allow multiple values for a key. Multiple values are clustered. Cursor is used to iterate through the keys.

5.2 Query Type 2

Q: Display the year in which Titanic movie was produced

Titanic: 1915
Titanic: 1943
Titanic: 1953
Titanic: 1984
Titanic: 1989
Titanic: 1993
Titanic: 1997
Titanic: 2012
Titanic: 2014

The query is answered by hashing for movie title into “movieYear.db”. All records with same key will be retrieved. Multiple entries with the same title is assigned a single key. Additional information may have been present in dataset that would have provided ways to differentiate the similar titles. The data in nine files that were included in the study, is insufficient to handle this issue.

5.3 Query Type 3

Q. Display the actors and actress of Superman:

Actors
Superman: Aldrich, Fred (I)
Superman: Alexander, Keith (I)
Superman: Alyn, Kirk
Superman: Anderson, Vass
Superman: Andrews, Harry (I)
Superman: Arnie, Harry
…..
Actresses
Superman: Alexander, Joan (I)
Superman: Bhatt, Urmila

Superman: Brown, Miquel
Superman: Carroll, Virginia
Superman: Case, Diane Sherry
Superman: Delany, Dana
Superman: Douglas, Sarah (I)
Superman: Easterbrook, Leslie
Superman: Forman, Carol (I)
…..

The query is answered by hashing the movie title into “movieActor.db” and “movieActress.db” and using a cursor to iterate through the rows.

5.4 Query type 4

Q. Display all locations where Director Speilberg, Steven has shot scenes:

Spielberg, Steven: 1941
1941: USA
Spielberg, Steven: A Timeless Call
A Timeless Call: USA
Spielberg, Steven: Always
Always: UK
Always: USA
Spielberg, Steven: Amazing Stories: Book One
Amazing Stories: Book One: USA
Spielberg, Steven: Amblin'
Amblin': USA
Spielberg, Steven: Bridge of Spies
Bridge of Spies: USA
Bridge of Spies: Germany
Bridge of Spies: India
…..

The query is answered by hashing the director name into “directorMovie.db”. For each movie that is returned, is hashed into “movieCountry.db” to retrieve the country locations.

5.5 Query type 5

Q. Display the movies actor Cruise, Tom produced and acted in:

Cruise, Tom: Adventurer's Club
Cruise, Tom: Ask the Dust
Cruise, Tom: Elizabethtown
Cruise, Tom: Hitting It Hard

Cruise, Tom: I Married a Witch
Cruise, Tom: Jack Reacher
Cruise, Tom: Jack Reacher: Never Go Back
Cruise, Tom: Luna Park
Cruise, Tom: Mission: Impossible
Cruise, Tom: Mission: Impossible - Ghost Protocol
Cruise, Tom: Mission: Impossible - Rogue Nation
Cruise, Tom: Mission: Impossible 6
Cruise, Tom: Mission: Impossible II
Cruise, Tom: Mission: Impossible III
….

The query is answered by hashing the actor's name into "producerMovie.db" and then for each movie that is fetched, is then hashed with the acotr's name to match both the key and value into "movieActor".

5.6 Query type 6

Q. Display number of movies of Producer DiFiglia, Tressa

1

The query is answered by opening a cursor and searching for the producer name in "producerMovie.db". The count of the search key is retrieved.

Chapter 6

Related work

6.1 Main Memory Limitation

Processing a query is fast when the database is loaded completely within the main memory. Previous works have focused on graph sizes that fit completely into main memory. But current growth rate of graph sizes is in sharp contrast to that of main memory sizes.

Distributed computing is an alternate way to overcome memory limitation on a single machine. Computing job is split into manageable chunks and computed simultaneously on multiple machines. Additional operating expense is involved to manage and coordinate distributed computing among commodity machines. Hence an alternate method for processing query on large graph database, notwithstanding memory limitation, is worthy.

6.2 Graph Representation

Existing approaches converted graph databases to structured relations that can be accommodated in main memory to facilitate query-processing. There are three drawbacks to this approach. First is the scalability concern. It places restriction on the size of the graph to be limited to the available main memory. Second is the cost of update. Changes in input data requires major effort to accommodate changes in relationships. Lastly, to answer the query requires gathering the relationship information from multiple expensive joins. Query and optimization techniques that are available may be implemented for faster processing. These gains may not be significant given the expensive joins and cost of updates.

6.3 Graph Query Representation

Graph querying is an important aspect of graph database. Querying language that allows easy specifications in query expression makes it endearing to users. Conventional database query-expression allows specification of entities, operators, joins and aggregates. Combination of various clauses help in finding specific instances. In keeping with this tradition, graph query should ideally allow entities, relationships, operators and aggregates to be expressed. Underlying representation of the graph database controls the design of query-specification characteristics. Existing work on query processing with graph databases were limited in their graph-query expression.

6.4 Existing Approaches

Graph Querying By Example (GQBE) [8] is a technique that accepts tuples of entities based on example tuple entities. It converts graph information to memory in hashed tables based on distinct edges. Query input is expanded to a Minimum Query Graph to capture the user's intentions. The maximum query graph is further processed to lazily generate lattice and prunes the irrelevant relationships. The results are sorted on a frequency information that is maintained and updated with every query. Top k results are displayed. Input query does not allow specification of relationships and operators.

GraphGrep [9] uses GLIDE query language for query specification that incorporates operators. *GraphGrep* builds hashed set of paths (database fingerprint) for each path of upto a certain length L. The hashed set of paths become unwieldy when L is increased beyond a limit. It returns the exact match of the query if such a path exists. It does not allow multiple query inputs. Forest of graphs is permitted, though the size of each input data graph is limited to 20 nodes or less.

Query processing using Subdue [10] is a memory based graph mining approach adapted to generate plans based on cost factor gathered from the generated catalog.

Partioned Graph Query Processing is an improvement over the previous technique. It achieves scalability by processing on partitions of graph that are loaded in turns into memory. Overhead is incurred in maintaining information in each partition. Multiple loading of partitions affects the performance of the results.

PathSim [11] generates network schema that describes the meta structure of the network

Chapter 7

Conclusion and Future work

7.1 Conclusion

In this thesis, we propose an approach for scalable conversion of textual unstructured data to NoSQL graph representation using Berkeley Database key-value store. Berkeley database is a high performance embedded database that provides hash storage type. Textual unstructured data is converted to a query friendly graph representation and redundantly stored in hash-indexed key-value store. Experiments are based on IMDb dataset. It is available as textual unstructured data in multiple files. Handful of files were picked from the entire dataset that contained the core information. IMDb dataset consists of data related to movie, TV series titles and their scene location, language spoken, people associated etc.

An entity type translates to a node-type. Entity value translates to a node in a graph data structure. Every node-type is assigned a pre-determined alphabet. Each node of one node-type is assigned a unique number. A predefined alphabet is suffixed with the unique number to generate a unique alphanumeric identification for a node over the entire data set.

The nodes and its node-types are identified and retrieved using patterns. The nodes are assigned unique ID. Two nodes and their relationship information forms a graph data structure. One of the node is designated as the key, and other node and its relationship becomes the value. This key-value pair is hashed into the relevant Berkeley Db index file based on relation. One graph file is generated for every directed relationship. Hence the implicit relationship information is not explicitly stored with the nodes' information. The nodes are interchanged to generate a reversed key-value pair. This

redundant information makes it easy to answer queries in multiple directions. Berkeley database stores hash files on secondary storage. Hence redundancy is affordable and primary memory size doesn't prove to be a constraint.

Many queries can be effectively answered in constant time with redundant information. Processing time for each file is improved by sharding the hash key-value store based on node-type and node value.

7.2 Future work

A query processing tool can be integrated to accept query input and display the results by adding a query interface for Berkeley DB index files.

The approach can be extended to include the remaining files in IMBD dataset

Also, one major shortcoming is that currently, we do not differentiate between different movies (or actors) with the same name, even though they are different entities. One approach would be to include the year of the movie as part of the key with name, to uniquely identify movies with the same name that were redone many times. This can also be used to identify and incorporate rank information to differentiate between duplicate and unrelated keys.

The graph files can be generated using Btree storage structure of Berkeley-DB instead of the currently used hash storage structure. This should allow retrieval of inexact matches allowing close or range matches.

We need to identify a way to incrementally process the added/deleted portions of the source and update the transformed representation of the graph. We assume that the graph does not change. If it does, we need to figure out a way to incrementally process the added/deleted portions and update the transformed representation of the graph.

The key-value graph representation can be implemented to larger datasets to develop a querying system.

References

[1] S. Batra and C. Tyagi, \Comparative analysis of relational and graph databases," International Journal of Soft Computing and Engineering (IJSCE), vol. 2, no. 2, 2012.

[2] https://en.wikipedia.org/wiki/NoSQL

[3] https://en.wikipedia.org/wiki/Berkeley_DB

[4] http://dblp.org/

[5] https://developers.google.com/freebase/

[6] https://en.wikipedia.org/wiki/IMDb

[7] Oracle/Berkeley20DB2012cR1206.2.32/docs/programmer_reference/intro_dbis.html

[8] Nandish Jayaram, Arijit Khan, Chengkai Li, Xifeng Yan, Ramez Elmasri. Querying Knowledge Graphs by Example Entity Tuples. In IEEE Transactions on Knowledge and Data Engineering (TKDE), 27(10): 2797-2811, October 2015.

[9] R. Giugno and D. Shasha, Graphgrep: A fast and universal method for querying graphs." in ICPR (2). IEEE Computer Society, pp. 112-115.

[10] A. Goyal. (2015) QP SUBDUE: PROCESSING QUERIES OVER GRAPH DATABASES. [Online]. Available: http://itlab.uta.edu/students/alumni/MS/Ankur Goyal/QPSubdue.pdf

[11] Y. Sun, J. Han, X. Yan, P. S. Yu, , and T. Wu. PathSim: Meta path-based top-k similarity search in heterogeneous information networks. VLDB, 2011.

Bibliographical Information

Jasmine Manoj Varghese was born in Gujarat, India. She graduated in Physics from Gujarat University, India in 1994. She has more than ten years of experience as a technical trainer and database administrator. She attained her Master's degree in Computer Science from University of Texas at Arlington in December 2017. She is passionate about data science.