A Novel Approach to Transform Relational Database into Graph Database using Neo4j

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Submitted By
Mehak Gupta
(801231017)

Under the supervision of:
Dr. (Mrs) Rinkle Rani
Assistant Professor

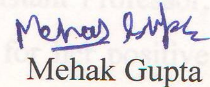
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2014

CERTIFICATE

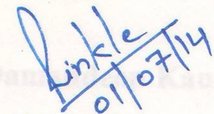
I hereby certify that the work which is being presented in the thesis entitled, "*A NOVEL APPROACH TO TRANSFORM RELATIONAL DATABASE INTO GRAPH DATABASE USING Neo4j*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. RinkleRani* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.



Mehak Gupta

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

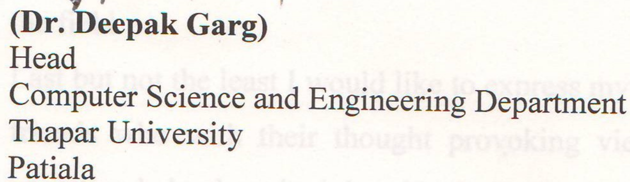


Dr. Rinkle Rani

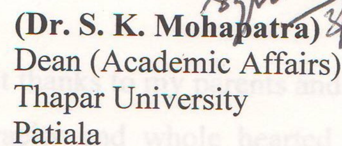
Assistant Professor

Computer Science and Engineering Department

Countersigned by



(Dr. Deepak Garg)
Head
Computer Science and Engineering Department
Thapar University
Patiala



(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

The successful completion of any task would be incomplete without acknowledging the people who made it possible and whose constant guidance and encouragement secured the success.

First of all I wish to acknowledge the benevolence of omnipotent God who gave me strength and courage to overcome all obstacles and showed me the silver lining in the dark clouds. With the profound sense of gratitude and heartiest regard, I express my sincere feelings of indebtedness to my guide **Dr. Rinkle Rani**, Assistant Professor, Computer Science and Engineering Department, Thapar University for her positive attitude, excellent guidance, constant encouragement, keen interest, invaluable co-operation, generous attitude and above all her blessings. She has been a source of inspiration for me.

I am grateful to **Dr. Deepak Garg**, Head of Department and **Ms. Damandeep Kaur**, P.G. Coordinator, Computer Science and Engineering Department, Thapar University for the motivation and inspiration for the completion of this thesis.

I will be failing in my duty if I do not express my gratitude to **Dr. S. K. Mohapatra**, Senior Professor and Dean of Academics Affairs in the University, for making provisions of infrastructure such as library facilities, computer labs equipped with internet facility, immensely useful for the learners to equip themselves with latest in the field.

Last but not the least I would like to express my heartfelt thanks to my parents and my friends who with their thought provoking views, veracity and whole hearted co-operation helped me in doing this thesis.

Mehak Gupta

(801231017)

Abstract

Nowadays, many companies rely on cloud services to meet their data storage requirements. Cloud does not support traditional relational databases because of scalability. Therefore, data needs to be migrated from relational to cloud databases. Graph Databases are one of the cloud databases which have been invented for fast traversal of millions of nodes that are interconnected via number of relations. They are becoming popular and efficient choice for storing and querying highly interconnected data. Traversal queries in SQL which require many joins to query relational database can be easily and quickly traversed using graph databases. Graph Databases have applications in many domains such as social network, organization management, banking, insurance, fraud detection, etc. Data migration is becoming a topic of interest these days because of increase in need of various data exchange formats. In this paper an approach has been suggested to convert relational database to graph database. An example database of auto-insurance has been used for the experiment. Experimental results have been presented to show feasibility of the proposed methodology. Query translation is done from SQL to Cypher query language. Query execution efficiency comparison is done on source and target databases.

Table of Contents

Certificate.. i
Acknowledgement……………………………………………………………… ii
Abstract…... iii
Table of Contents.. iv
List of Figures.. vi
List of Tables.. vii

1. Introduction………………………………………………….……......... 1
1.1. Classes of NoSQL…………………………………….………......... 1
1.1.1. Key/Value Stores..……………....………………..…….. 1
1.1.2. Document Databases ……………………………...………….. 2
1.1.3. Column Oriented Databases….......……………….……… 3
1.1.4. Graph Databases……………..…………………….............. 4
1.2. Features of Graph Databases……………..…..........……………… 4
1.3. Relational vs Graph Databases………………………..……...…… 6
1.3.1. Scalability………………………………..………….…..... 6
1.3.2. Transactional Properties……………………….…………...... 7
1.3.3. Data Model…………………………………………….……...... 10
1.3.4. Database Schema…………………………………………….... 11
1.3.5. Query Languages……………………………………….……….. 12
1.4. Applications of Graph Databases.……………………...………... 12
1.4.1. Retail Market……………………………………………… 12
1.4.2. Data Analytics……………………………………………....... 13
1.4.3. Social Graphs………………...…………………………….. 13
1.5. Neo4j…………………………………………………………….. 14
1.6. Cypher Query language…………………………………………. 15
2. Literature Review...…………………………………….………… 19
2.1. NoSQL Data Models……………………………………………….. 19
2.2. Neo4j……....………………………………………………….. 21
2.3. Cypher Query Language………………………………………....... 22
2.4. Data Migration Approaches……………………………………... 23

3. Research Problem.. 25
3.1. Problem Statement.. 25
3.2. Research Gaps.. 25
3.3. Objectives... 26
3.4. Research Methodology.. 26
4. Transforming Relational to Graph Database..................................... 27
4.1. Preliminaries... 27
4.2. Types of relationships in Relational Database................................ 29
4.3. Converting data from relational to graph database........................ 31
5. Implementation and Results.. 39
5.1. Implementation Environment... 39
5.2. Database Design... 39
5.3. Experimental Results... 41
5.4. Query translation methodology.. 44
5.5. Performance comparison of SQL and CQL..................................... 45
6. Conclusion and Future Scope.. 50
6.1. Conclusion.. 50
6.2. Summary of Contributions... 50
6.3. Future Scope... 51
References.. 52
List of Publications... 56

List of Figures

Figure No.	Description	Page No.
1.1	Representation of Key-Value Data Store……………………………	2
1.2	Representation of Document Data Store…………………………….	3
1.3	Representation of Column family Data Store………………………..	4
1.4	Graph Model………………………………………………………	6
1.5	CAP Theorem……………………………………………………..	9
1.6	Property Graph…………………………………………………….	11
1.7	Basic friend-of friend graph……………………………………….	14
2.1	CAP Properties in various databases……………………………...	19
2.2	NoSQL Data Models……………………………………………...	20
2.3	Evolution of database models……………………………………..	21
4.1	An example Relational Database R……………………..………...	28
4.2	Transforming 1:N Relationship from relational to graph database….	29
4.3	Transforming N:M Relationship from relational to graph database....	30
4.4	Flowchart……………..…………………………………………	33
4.5	Graph Database G……..…………………………………………	38
5.1	ER Diagram of IMDb subset Database……………………………...	40
5.2	Number of nodes of label “Movie”………………………………….	41
5.3	Number of nodes of label “Person”………………………………….	42
5.4	Number of nodes of label “Character”………………………………	42
5.5	Number of nodes of label “MovieInfo”……………………………...	42
5.6	Number of nodes of label “Role”……………………………………	43
5.7	Number of edges with name “Cast”…………………………………	43
5.8	Number of edges with name “MovieInfo”..…………………………	43
5.9	Comparison of SQL and CQL on the basis of execution time ……...	48

List of Tables

Table No.	Description	Page No.
1.1	Various clauses used in Cypher query language………..	16
4.1	Schema details of example relational database R……….	28
5.1	IMDb Database subset used for implementation………..	40
5.2	Five queries with the relations involved in their joins…..	46

Chapter 1 Introduction

Relational databases have been in existence since 1970s [1] and are the database of choice for most of the applications those require to store and retrieve information from large amount of data. Due to the prolonged existence of these databases, they have matured very well. But, these databases were not designed to support horizontal scalability. They were not designed to efficiently handle sparse and very large data. But in today's scenario of increasing data, there is a growing need for scalability of databases [2]. Scalability can be achieved in two ways – vertical scalability and horizontal scalability. Vertical scalability means to scale up. This is achieved by increasing resources to a single machine. Horizontal scalability on other hand is called scaling out. This can be achieved by adding commodity servers to the existing node. Vertical scalability is expensive as compared to horizontal scalability. Traditional databases which were designed to work on single node can be scaled vertically. But vertical scalability is limited and expensive. With increasing data, vertical scalability is not an efficient option and in some cases not feasible choice. Recently evolved non-Relational Databases also called as NoSQL databases use horizontal scalability [3]. These data stores can be scaled by increasing commodity servers or by increasing cloud instances. There are many other advantages of NoSQL databases over Relational Databases such as NoSQL databases are schema-less, highly scalable, etc. These NoSQL databases have mainly four classes – Key-Value pair, Document, Graph and Column-oriented. Each class of NoSQL databases have its own specific properties to suit different types of data and thus different storage and retrieval requirements.

1.1. Classes of NoSQL

1.1.1. Key/Value-stores

Key-value data stores use a simple data model with a single unique key-value index for all data. This unique key is used for all the operations to be performed on data such as delete the key, update the entry with the given key, insert new key. Key-value data stores also allow the application to store its data in a schema-less way [4]. The data type stored can be from any programming language or of object type. These data stores provide scalability over consistency. Key-value data stores have been in

existence for a long time e.g. Riak [5], voldemort [6], Berkeley DB [7] but, this class of NoSQL data stores that are emerging recently are mainly influenced by Amazon`s Dynamo.

Figure 1.1 shows simple representation of key-value data stores. It shows how data is stored in the form of keys and values, where keys can be simple objects and values can be sets, lists [8]. A row key is used to uniquely identify each row.

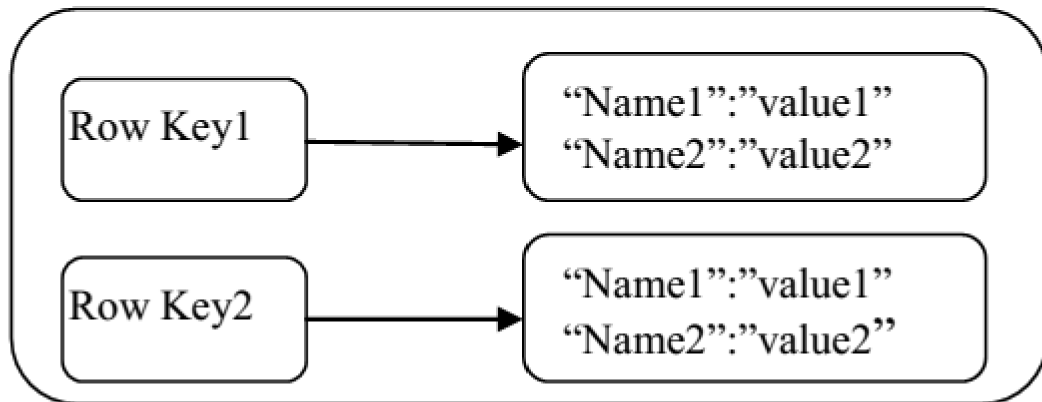


Figure 1.1: Representation of Key-Value Data Store

1.1.2. Document Databases

Document database contains semi-structured data. In comparison to Relational Databases, document data stores can have any number of fields in each tuple. This allows application programmer to have varying number of fields of varying length in each tuple without wasting space due to empty fields. These stores use unique key index to identify documents. Data is stored in denormalized form in which related data is stored in single document [9]. Like other NoSQL data stores document data stores also does not support ACID transactional properties. Examples: MongoDB[10], CouchDB, etc.

Figure 1.2 represents document databases.

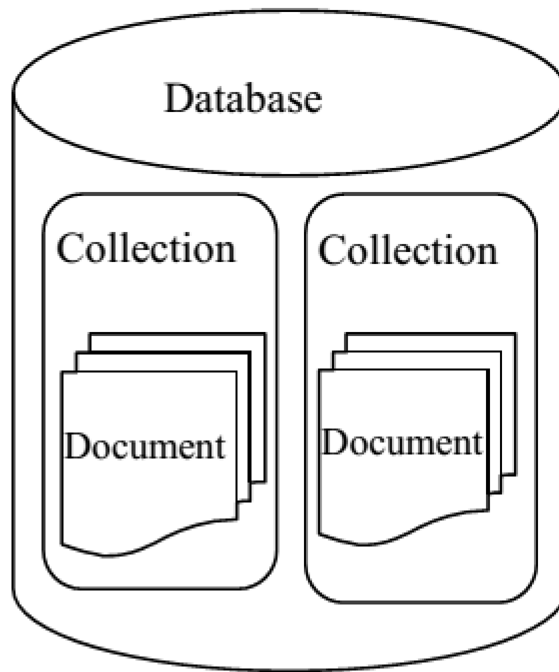


Figure 1.2: Representation of Document Data Store

1.1.3. Column-Oriented Databases

Column oriented data stores also called as wide table data stores have been deigned to handle large amounts of data. They are excellent in handling sparse data. These data stores contain column families. Key indexes are used to identify each column family. Column family consists of many columns. Each column family defines how data is to be stored on each disk. Data is stored in a way that columns are stored contiguously rather than rows [11]. This leads to better performance when very few columns are needed out of the large number of columns in the result set. All columns belonging to single column family are stored on same file. The main inspiration for column-oriented data stores is Google's BigTable [12]. HBase is one of the most prominent column oriented database. It is Hadoop based database [13]. It uses a data model very similar to Google's BigTable.

Figure 1.3 represents column-oriented databases. In comparison to Relational Database, column family in column database corresponds to table in Relational Database, row key to primary key. Columns name/key and value in column database to column name and value in Relational Database.

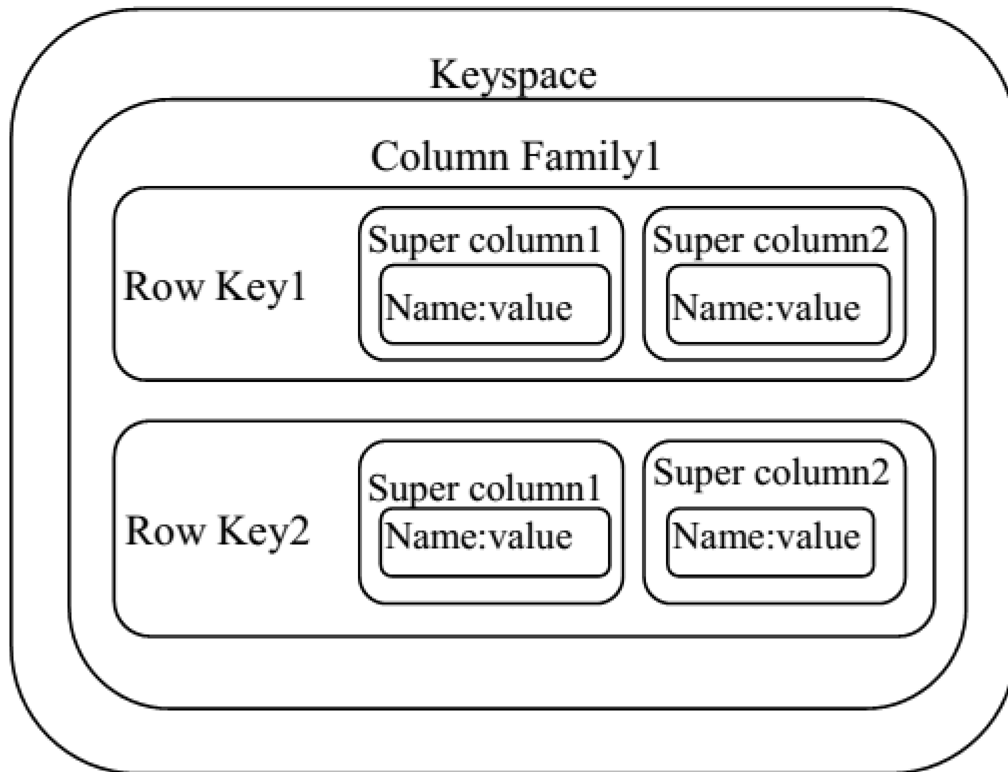


Figure 1.3: Representation of Column family Data Store

1.1.4. Graph Databases

Graph Databases have been invented for fast traversal of millions of nodes that are interconnected via number of relations [14]. Social network websites like facebook and twitter produce several terabytes of data within a month. They contain highly interconnected data. To traverse such data Graph Databases are used. Graph Databases provide flexible model to distribute data over many servers. This property provides high scalability for growing data. In graph model every node has its related node directly connected to it. This eliminates the need of any index lookup or table joins as in RDBMS. Graph model based data stores are backbone of social networking websites. Example, Neo4j [15], etc.

1.2 Features of Graph Databases

Mathematics have been an integral part of Computer Science. Many concepts such as cryptography, automation and other simple concepts of mathematical logic and Boolean algebra closely couple the two mathematics and computer science. Graph theory is another concept that has been used in Computer Science. Many data

structures such as organizational hierarchy, flow diagrams, social network are represented by graphs.

Where graphs are so extensively used in software development, they are generally forgotten about when there is the need to store and query real-world data. Data is generally stored in Relational Databases which dominate the market since 1970s. The data is normalized to the extent that it no more looks like the actual data. For example, there is the need to store movie database containing movies, actors and genre. For this, different Relational tables have to be created to contain movies, actors and genre information. Than another join table has to be created to map actors to movies and movies to genre. In the end there will be 5 Relational tables to store simple data which actually represent a graph with nodes and relations between them.

Graphs in Graph Databases contain information on both nodes and edges. Edges define the relation between two nodes and they have their own properties to provide information about the type of relation. Similarly, nodes too have their own properties.

Graph models are used to represent data that is best represented in node-relationship format. Data which is highly connected and has undetermined number of relationships, which have their own unique information to reveal, is represented in graph models [16]. Data in multiple domains can be represented as graphs such as Semantic Web, images [17], social network, videos [18] and bioinformatics.

Due to the basic nature of graphs the cost to travel between two nodes is calculated by number of hops between those nodes. This nature of graphs makes the traversal cost from one node to remain constant irrespective of the data type used to store the information [19]. This low moving cost is the main reason for high performance of graph data models. Cost of moving from hop to hop remains same even if number of nodes increases.

Graph Database can store semi-structured data, which is closed to the real world data. It supports schema less data [20]. This makes it easy and simple to insert, delete or update relations between nodes because they are not part of schema rather they are actual data.

Recently, Graph Databases have gained a lot of importance in social networks. Social networks contain large amount of data. Social networks like facebook, twitter produce several terabytes of data a month. This data is highly interconnected and would

require lots of joins to query data if, stored in Relational Databases. Whereas, in Graph Databases this highly connected data is represented by graphs with each related nodes adjacent to each other. Hence, querying such data is fast in graph models.

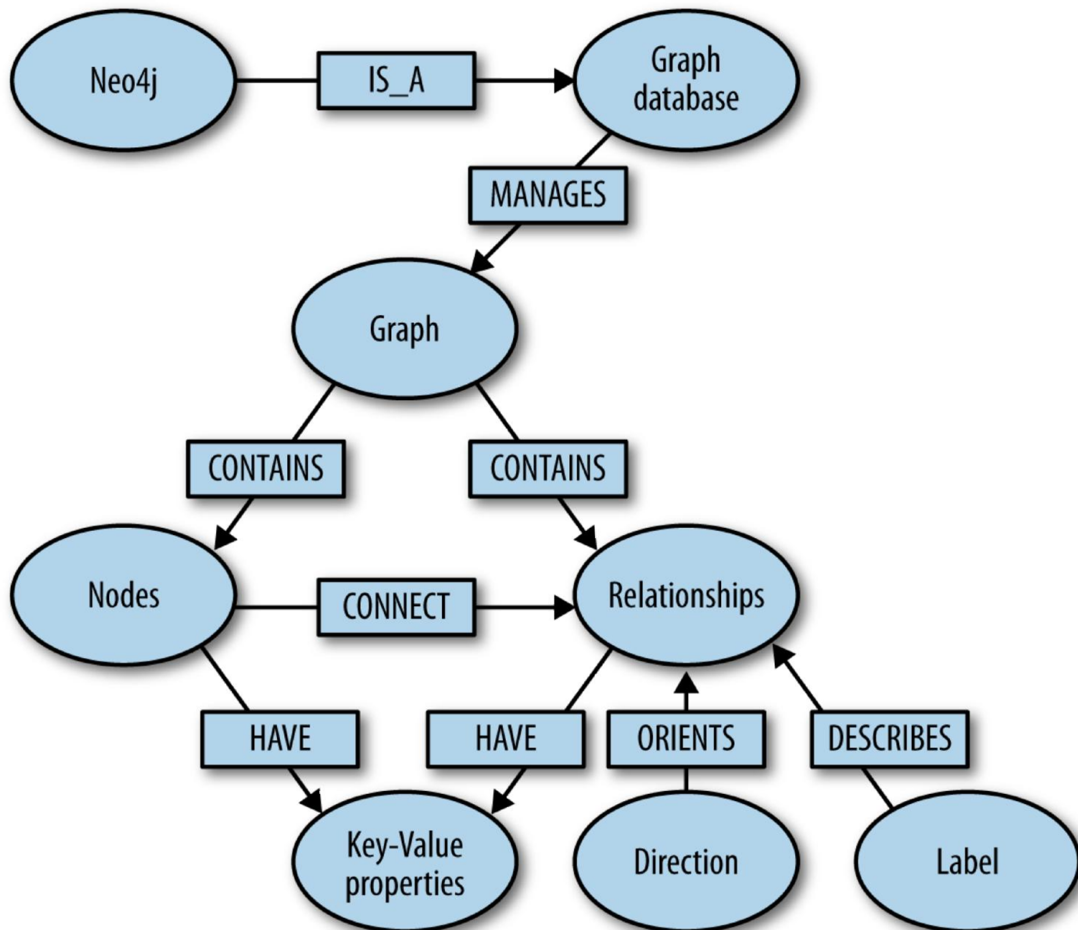


Figure 1.4: Graph Model[15]

1.3 Relational vs Graph Databases

1.3.1 Scalability

Scaling is an important feature which determines the usability of database. In today's scenario where data is growing every day, there grows the need to scale our databases. Scaling is of two types - horizontal and vertical scalability. Vertical scalability is achieved by increasing the capacity of one machine. It is an expensive option to scale

the data. Horizontal scalability is the option that is used to scale the dataabses. It can be achieved through two technologies – Replication and Sharding.

- Replication- Replication means that data is replicated on many nodes. This makes the data robust. If one node fails the lost data can be replaced with another node which has the same data. It also increases the performance of read operations because load balancer distribute the read operations over many nodes. But, for each write operation data has to be updated on all the replication nodes before it return to its new stable state. Thus, this technique slows down the write operation. This consistency issue is problem in Relational Databases. Whereas, in Graph Databases CAP theorem is followed. As discussed before CAP theorem allows eventual consistency to achieve high availability.

- Sharding – Sharding means data is splitted over many shards. The data is distributed over many nodes using some hash function applied on the primary keys. It means that data related to one entity may not be present on one machine rather distributed over many nodes. Sharding is very flexible technique in which nodes can be added when data grows and can be removed when data decreases without affecting the application.

  In Relational Databases, sharding becomes a problem when joins are required between two relations. In join operation there are two sides – left and right. All the data matching the left has to be retrieved from the right. This requires communicating with many nodes. Because of this non-Relational Databases such as Graph Databases does not support joins.

1.3.2. Transactional Properties

Till date the primary benchmark against which databases are measured is ACID transactional properties of the system. But the NoSQL databases that are being discussing here do not provide ACID properties and without ACID properties the reliability and quality of database systems is suspect. ACID stands for Atomicity, Consistency, Isolation and Durability.

In RDBMS, atomicity, consistency and isolation are implemented. Consistency in RDBMS is achieved by a central locking system. In this data is locked until it reaches

its stable state. But in distributed environment this central locking system becomes an overhead for performing transactions. Thus these new NoSQL databases including Graph Databases go for eventual consistency which is weaker type of consistency. They drop strong consistency to achieve high availability. This leads to systems know as BASE (Basically Available, Soft-state, Eventually consistent).

In 2000, Eric Brewer presented his keynote speech at the ACM Symposium on the Principles of Distributed Computing and CAP Theorem was born [21]. CAP theorem suggests that you cannot adopt all the constraints consistency, availability and partition tolerance at one time. You can adopt only two out of the three constraints given by CAP theorem.

The CAP acronym stands for:

- **Consistency** means that in distributed environment all the servers will have same data. Data will be accessible to all readers and writers of the system in a shared mode regardless of any update operations performed by writers. Same data can be from any node by anyone.

- **Availability** means that system should be designed in such a manner that it should always return data even if a node in the system goes down. It should return data even if it is not the latest or consistent data across the system. Systems with high availability have the ability to continue read/write operations even if nodes in a cluster crash.

- **Partition Tolerance** means the system have the ability to operate as whole even if individual servers are up but can’t be reached. Partition tolerance of the system guarantees the operation of the system even if network failures prevent the servers from communicating with each other. This is important in scenario where data is partitioned and distributed over many servers. It is also useful where nodes are added or deleted dynamically for maintenance or other purposes.

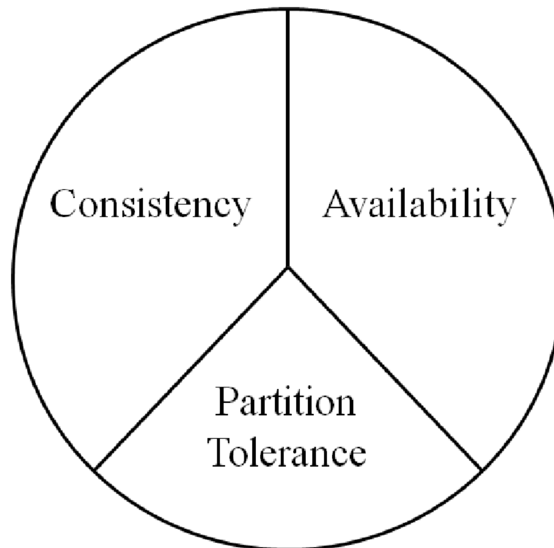


Figure 1.5: CAP Theorem[21]

Now, Brewer alleges that at most two of the three characteristics can be implemented at one time in a "shared-data system"

Dealing with CAP

Drop Partition Tolerance

In this case, one can perform read/write operations on data as long as all nodes are online and can be sure that data is consistent. Problem occurs when you create partition among nodes. Due to partition data goes out of sync. One way to avoid partitioning in this scenario is to put all data on one machine or in one atomically failing unit. Scaling of the application is the obvious limitation of this scenario.

Drop Availability

In this case, availability is dropped to achieve consistency on partition tolerant systems. In such systems, when any update operation is performed by writer than data is made consistent throughout by making it unavailable for some time. System is made unavailable for the time updated data is being broadcasted to all the nodes. Controlling consistency this way is difficult over many nodes, because re-available nodes need logic to handle their comeback gracefully.

Drop Consistency

In this scenario, consistency cannot be guaranteed either during or after partition. Due to availability and partition tolerance of the system, nodes remain online even if they cannot communicate with each other. But data will be resynchronized only after partition is resolved. These systems achieve consistency eventually.

Inconsistencies generally do need much effort to be solved. Systems generally do not need continuous consistency. For example, if two customers are buying a same item from online website than second customer's order will become back-order. As long as customer is informed of this denial of purchase, there is no side-effect of inconsistency in the application.

1.3.3. Data Model

The Relational model

The Relational model is designed on the principle concept of normalization. Normalization allows Relational model to store any data without redundancy and loss of information. Once the data model is ready, data can be inserted and manipulated using very strong structured query language SQL.

But the problem hits when deep SQL queries are implemented that requires spanning of many table joins. Highly interconnected data such as networks, social data which can be stored using large number of joined tables, is having problems because of expensive join operations. The join operations have very low performance and are not scalable with growing number of tuples. Other limitations of Relational models include inefficiency in storing semi-structured data and sparse data. Such data is stored in large tables where many of its columns are empty.

Graph Model

There are many different types of graph models. In Graph Databases property graph model is implemented. Figure 1.6 shows basic structure of property graph.

This graph contains nodes, edges and properties. It is directed, labelled and attributed multigraph. Directed graph has its edges with fixed direction. Labelled graph means

that its edges have labels. Attributed graph means that it has variable list of attributes for all nodes and edges. Multigraph allows more than one edge between two nodes.

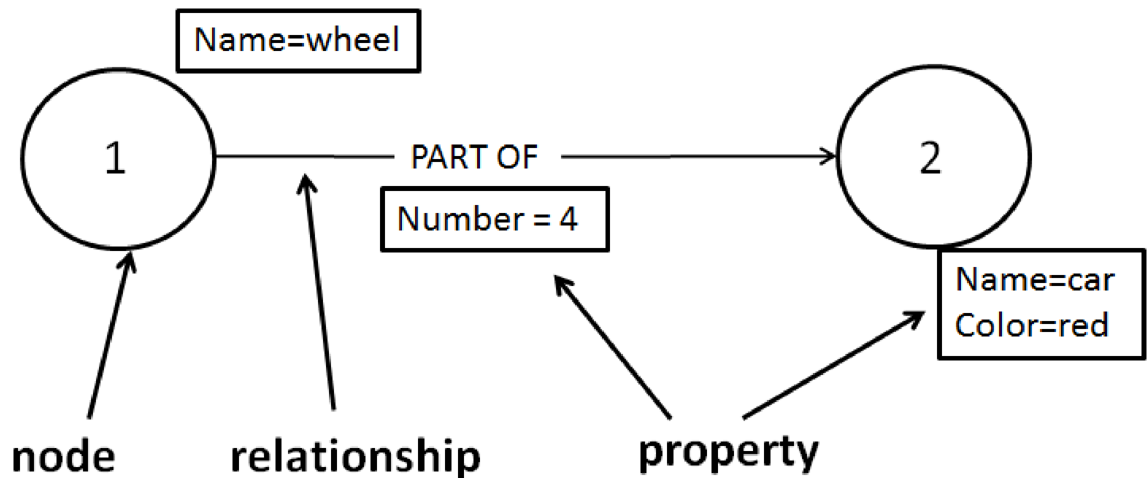


Figure 1.6: Property Graph[22]

1.3.4. Database Schema

Relational Database supports very rigid schema. An application which uses Relational Database has to make decision about its schema first before inserting the data. That schema is consistent throughout the application. All the tuples in a relation follow the same schema i:e all the tuples have same columns. This leads to sparse data when most of the columns remain empty.

Graph Databases follow flexible schema [23]. Each node and edge in a Graph Database can define their own properties. There is no particular schema to be followed by the nodes and edges of the Graph Database. So if a node or an edge in a Graph Database have null values for some properties they do not need to allocate space for those properties.

1.3.5. Query Languages

Relational Databases use structured query language (SQL) for insertions, deletions, updates and querying the data. SQL is declarative language. It is based upon Relational calculus and Relational algebra. It is in existence for a long time and is consistent throughout its implementation.

Graph query languages are based on the fundamental concept of graph traversal [24]. The strength of graph query languages is measured on their ability to efficiently traverse the graphs. Traversal means walking through the graphs from node to node via connecting edges. Information from graphs is retrieved by traversing them.

Most commonly used graph query languages are Cypher and Gremlin. Cypher is also declarative language inspired from SQL. Gremlin is domain specific language for graphs. It implements traversals through piping.

1.4. Applications of Graph Databases

1.4.1. Retail Market

On 18th, March, 2014 Neo technology – the creator of Neo4j, announced that big retailers like eBay and Walmart are using Neo4j for their critical business projects [25]. Ebay is using Neo4j to improve the performance of their same day delivery service. Walmart is using Neo4j to understand the behaviour of online shoppers and able to optimize the sale of major product lines in core market.

With the explosive growth and addition of many new features, the company needed to rebuild its service platform. Its old MYSQL solutions were too time consuming and its complex queries were too large to maintain. The ebay development team knew that Graph Databases could solve their problem of performance and scalability.

"We found Neo4j to be literally thousands of times faster than our prior MySQL solution, with queries that require 10-100 times less code. Today, Neo4j provides eBay with functionality that was previously impossible," said Volker Pacher, Senior Developer at eBay [25].

Walmart uses Neo4j to understand the behaviour of buyers. Today the large number of reviews provided by buyers and their tendency to consult social media before

buying a product makes a huge impact on product sales. Neo4j helps to understand online shoppers and the connection between buyers and products, which provides them with a real time tool for product recommendations.

1.4.2. Data Analytics

With that increase in data produced by machines and people, there is growing need of new analytical capabilities. Graph Databases are gaining popularity because of the amount of data they aggregate and analyse. Graph Databases contain nodes and edges. Nodes represent things like people, products. All nodes have some properties which describe these nodes. Edges act as relationship between nodes. Edges connect various nodes defining the type of relationship between two connecting nodes. Data is analysed by capturing relationship pattern between nodes.

Due to the invention of new wearable technologies like Google Glass which uses many sensors to record data, the demand for Graph Databases is increasing. Hence, it will be important to collect data from various sensors that are in house or cars and relate them to make sense out of it. There is also the need to analyse increasing amount of data such as received from resources like medical records, contracts, etc. It is the intersection with APIs and the patterns that come from Graph Databases and text search capabilities that Apigee sees as a long-term opportunity with InsightsOne.

1.4.3. Social Graphs

Many of the largest social networking websites are using Graph Databases to maintain their highly interconnected data.

Social graphs are becoming more and more important for solving many problems which require accuracy in recommendations such as adding friends, online dating or online shopping, etc.

SNAP: Many dating sites are using Graph Databases to recommend people in their extended social graph. To avoid going out with total strangers they recommend friend of friend or friend of friend of friend. Other than social graph they also use passion graph which relates interests of two persons and location graph. Figure 1.7 shows a basic friend-of-friend graph.

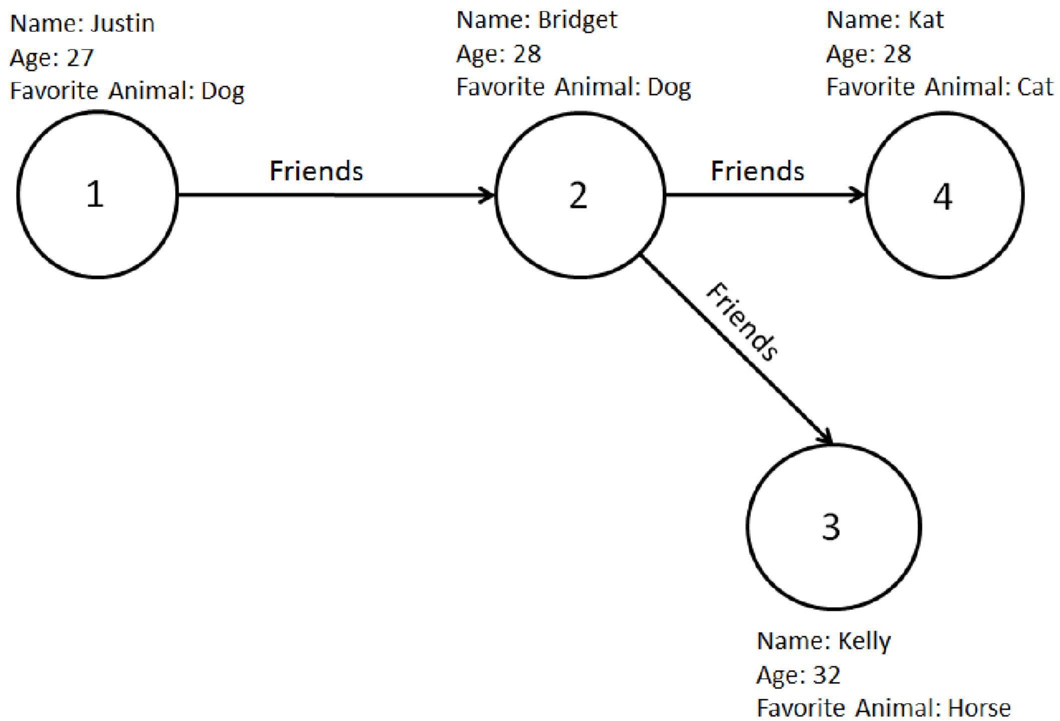


Figure 1.7: Basic friend-of friend Graph [22]

Glassdoor: It uses the graph of billions of users and their friends. It maintains all the data about companies, jobs and job seekers. Snap and Glassdoor both reported the high improvement in their recommendations by using Graph Database to analyse the network of highly connected data. By finding and making better use of networks, many different types companies are breaking new ground with respect to intelligent real-time analytics.

The beauty of Graph Databases is that they not only store information on nodes but also have information about the relationships among the data. This capability of Graph Databases helps them to answer many sophisticated questions in minimal time.

1.5. Neo4j

Neo4j is an open source NoSQL Graph Database. It is a Java-based Graph Database. Neo4j version 1.0 was released in February, 2010. Latest version of Neo4j which has been released in December, 2013 is Neo4j-2.0. This version produced one of the most significant modification in Graph Databases, the user interface which has been completely rewritten. This new UI is clean and contain many helpful reference topics.

Cypher query editor has many features such as visualizations, tabular representation [26], export features and drag-and-drop support.

Any software application can have reliable graph storage added to it. Graph Databases are highly scalable. They can be scaled from single server to multi server high availability installations. Scaling has very little impact on performance of graphs. Whether creating new application or for maintenance purpose of old applications, Neo4J can be easily scaled up. It is only limited by available physical resources. A single server instance of Graph Database can store millions of nodes and relationships. Graph Databases outstand in comparison to other databases when storing and querying highly connected and inter-related data. All the related nodes are adjacent to each other and graph querying is performed by traversing the nodes along the connected edges which represent relations among nodes. Hence, millions and billions of nodes can be traversed in few seconds. These traversals are performed through table joins in RDBMS, which results in degradation of their performance due to overhead of joins of large data. Moreover, RDBMS has to have structured data which is far from the way data exists in real-world.

1.6. Cypher Query Language

Cypher query language is declarative query language. It is a graph query language that allows efficient and expressive retrieval and update query syntax without the need of writing the traversal paths for graph model in code. The results obtained are expressive and can be customized according to the need. It is still evolving which means there can be some major changes in its syntax in future. It also suggests that it has not undergone the rigorous performance testing to the extent other Neo4j components has gone.

Cypher query language is designed in a manner to be suitable for both programmers and operations professionals. The motivation behind Cypher query language design is to make the graph querying simple and to make complex tasks possible [27].

The design of Cypher query language is influenced by a number of different parameters. It's design is based upon already existing expressive query languages. Many keywords such as ORDER BY and WHERE are taken from SQL. The approach behind pattern matching is borrowed from SPARQL. Since cypher is a

declarative query language, its main focus is on the clarity of the syntax of the expressions. Its syntax focusses more on what information to retrieve from graph, rather than on how to do it. This is in contrast to scripting languages, imperative languages like Java and the JRuby Neo4j bindings. The concern of optimizing Cypher queries is in its implementation detail, which is not exposed to the user.

Cypher query language's also affected the graph model of Neo4j 2.0. Cypher is more declarative and concise in 2.0. The new feature of Neo4j 2.0 is its extended support for labels and indexes. Now nodes of the graph can be labelled and its properties can be used as indexes. This feature improves the performance of cypher queries when graph size and nodes increases. It makes queries faster and easy [26].

Table 1.1 shows the clauses used in Cypher query language.

Table 1.1: Various clauses used in Cypher Query Language

Cypher Language Clauses	Explanation
START	Starting points in the graph, obtained via index lookups or by element IDs
MATCH	The graph pattern to match, bound to the starting points in START
WHERE	Filtering criteria
RETURN	What to return
CREATE DELETE	Creates nodes and relationships Removes nodes, relationships and properties
SET	Set values to properties
FOREACH	Performs updating actions once per element in a list
WITH	Divides a query into multiple, distinct parts

Read-Only Query Structure
START me=node : people (name = 'Andres') [MATCH me – [:FRIEND] -> friend] WHERE friend.age > 18 RETURN me, friend.name ORDER BY friend.age asc SKIP 5 LIMIT 10

START	*meaning*
START n=node (id, [id2, id3])	Load the node with id into n
START n=node : indexName (key = " value ")	Query the index with an exact query and put the result into n Use node_auto_index for the auto-index
START n=node : indexName (key = " lucene query ")	Query the index using a full query and put the result in n
START n=node (*)	Load all nodes
START m=node (1) , n=node(2)	Multiple start points

MATCH	***meaning***
MATCH n → m	A pattern where n has outgoing relationships to another node, no matter relationship-type
Match n -- m	*n* has relationship in either direction to m
MATCH n – [:KNOWS] → m	The outgoing relationship between n and m has to be of *KNOWS* relationship type
MATCH n – [:KNOWS | LOVES] – m	*n* has *KNOWS* or LOVES relationship to *m*
MATCH n – [r] → m	AN outgoing relationship from *n* to *m*, and store the relationship in r
MATCH n – [r?] → m	The relationship is optional

RETURN	***meaning***
RETURN *	Return all named nodes, relationships and identifiers
RETURN *expr AS alias*	Set result column name as alias
RETURN distinct *expr*	Return unique values for *expr*

2.1. NoSQL Data Models

With the growth in internet and cloud computing there is a need for databases which can support scalability and store large amount of data efficiently. New emerging applications have several new requirements such as high scalability, high availability, efficient storage and access of big data. To fulfil these several needs a new type of databases were developed which was named as NoSQL. These databases are different from traditional Relational Databases and are thus called NoSQL i:e "Not only SQL". Various data models are available in NoSQL databases. Unlike traditional Relational Databases which were designed to support ACID transactions, these new data models follow CAP theorem. Mainstream data models are – Document, Key/Value, Graph and Column-oriented. Figure 2.1 shows which data models support which properties of CAP Theorem.

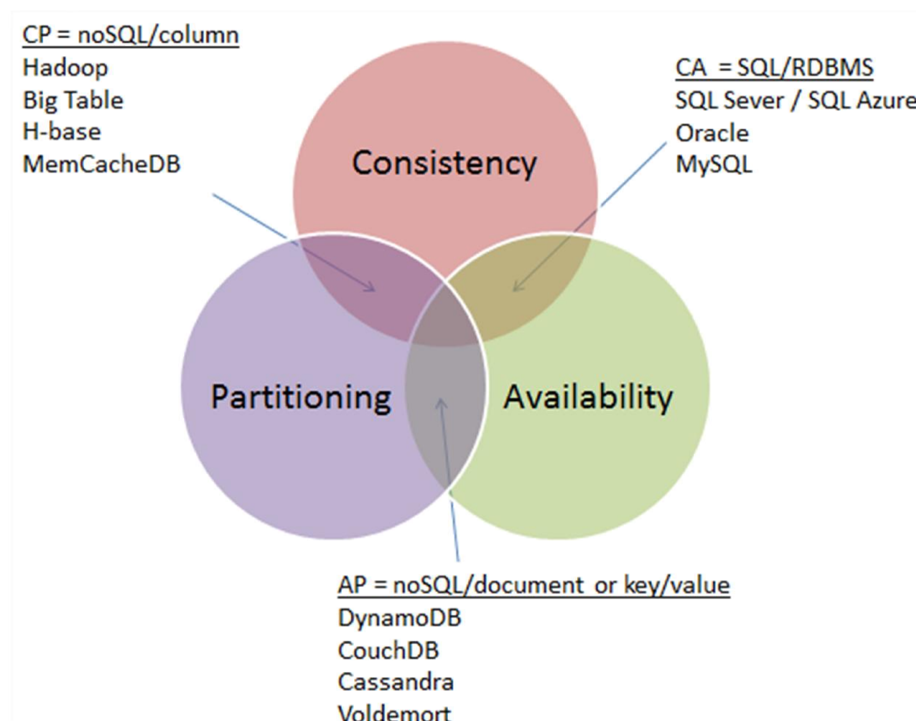


Figure 2.1: CAP Properties in various databases

Figure 2.2, shows the classification of four mainstream NoSQL databases based on the data complexity and data size. Data size is represented along vertical axis and data complexity is represented along horizontal axis.

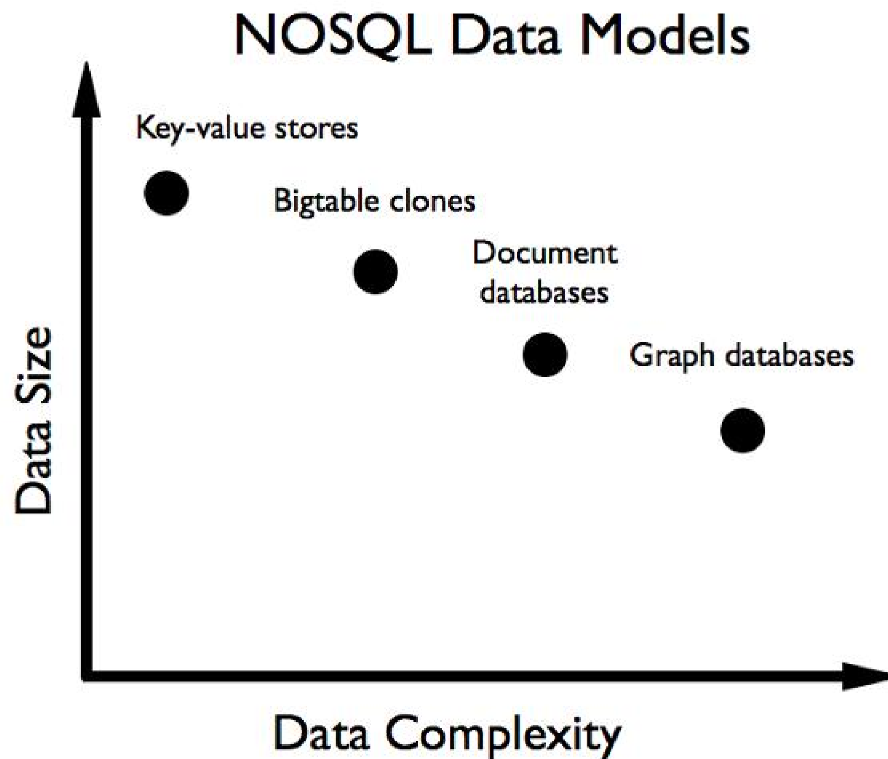


Figure 2.2: NoSQL Data Models[28]

Graph Database models have their schema modelled as graphs and their data manipulation queries are represented by graph-oriented operations. Their influence faded with the invention of other database models such as geographical, XML and spatial. Recently due to the increasing need of representing the data via graph models, has increased the importance and need to study graph data models. Renzo Angles [29] presented a work on comparison between various graph data models based on their features such as data storing, data structure, constraints and query languages.

Figure 2.3 shows the evolution of database models.

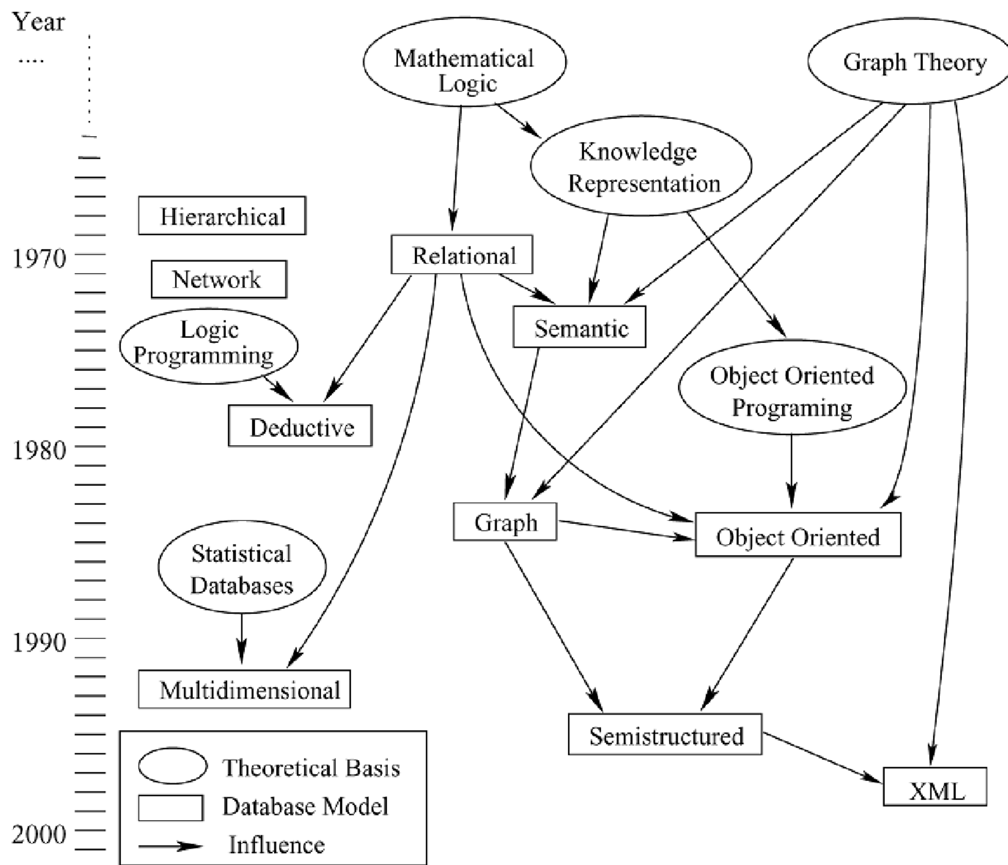


Figure 2.3: Evolution of database models. Rectangles denote database models, arrows indicate influences, and circles denote theoretical developments. A time line in years is show in the left [30].

2.2 Neo4j

Neo Technology was formed in 2007. One of the major offering of Neo Technology is Neo4J. Neo4J is an open source NoSQL Graph Database. It is a Java-based Graph Database. Graph Databases can search and explore highly connected and inter-related social network. Neo4j (Graph Database) can solve problems of querying highly connected by replacing joins of large tables in RDBMS by simple graph traversals.

Neo4j is an open source project. It has two editions - community edition and an Enterprise edition. Community edition runs fully featured graph. But this graph could

be run on one node only. Whereas, Enterprise edition provides clustering across many, many machines.

Neo4j version 1.0 was released in February, 2010. In 2012 attempts were made to make an already solid and reliable product even more reliable, with a 1.6 release in January, 1.7 in April, and 1.8 in October. The result of improvements in these releases is a much stronger and reliable database. The 1.9 release of Neo4j released on (2013-05-13), builds upon the previous 1.8 releases and introduced extensive modifications and improvements in stability and performance. In addition to performance improvements, Neo4j 1.9 Edition also introduced auto-clustering support. This dramatically simplified deployment and configuration for high demand production use.

Latest release of neo4j has been released in December, 2013. Neo4J 2.0 which has been in the development process since early 2013 has been finally released after a lot of engineering effort. This version produced one of the most significant modification in Graph Databases. The user interface has been completely rewritten. This new UI is clean and contain many helpful reference topics. Cypher query editor has many features such as visualizations, tabular representation[26], export features and drag-and-drop support.

2.3. Cypher Query Language

Cypher query language's way towards more complete cypher has also affected the graph model of Neo4J 2.0. Cypher is more declarative and concise in 2.0. The new feature of Neo4J 2.0 is its extended support for labels and indexes. Now nodes of the graph can be labelled and its properties can be used as indexes. This feature improves the performance of cypher queries when graph size and nodes increases. It makes queries faster and easy [27].

These labels are just like names given to nodes to identify them. It creates a subset of graph in the whole database. Indexes are used to further identify a node using the property of the node which has been used to create the index. There can be many indexes and labels for one node.

Two new Cypher clauses have also been introduced. “Optional Match” is used if you don’t have to match everything in the pattern. Optional part in this clause may be missing and can return null if not found. “Merge” is another new clause added to Cypher. With merge clause if a node pattern matches with already existing node than that node is returned otherwise a new node is created.

2.4. Data Migration Approaches

With the advent of cloud based services many companies especially small companies rely on cloud services where you pay as-per-need basis. Most of the cloud platforms support data scalability and thus do not support Relational Databases. They support NoSQL database which support scalability of data and semi-structured data. Thus there grows the need for data migration from Relational to cloud databases. Data migration has the topic of research for a long time now. But these days where choices of data storage are increasing at a very fast pace, there grows the need for data migration between these available databases.

There has been many algorithms proposed for data migration and schema-mapping between Relational to non-Relational and between non-Relational to non-Relational. While migrating from Relational to non-Relational the biggest challenge is to transfer schema information to schema-less databases. For schema transformations where Relational Database is the source database, three types of relations are considered - one-to-one, many-to-one or one-to-many and many-to-many. Relational Databases have these relations managed by foreign keys. [31]. Column-families in case of Column-oriented databases and nodes in case of Graph Databases are considered joinable according to their foreign key relation in Relational Databases. For Graph Databases which are mainly used for traversal, schema paths are generated from foreign key relations in Relational Database. Many approaches have been suggested for database translation but there is no focus on data migration and query mapping between Relational and Graph Databases [32].

Majority of the data available on semantic web [33] is stored in RDF format. AS shown in [34] most of the current web data is stored in RDB. Thus to make data available on semantic web, there is the need to bridge gap between RDB and RDF. Recent research has shown that semantic web technologies are better that web,

particularly in the case where data has to be integrated and exchanged among different sources [35,36,37]. So, many organizations needed to convert their data from Relational to RDF to make it available on semantic web. This increased the need to convert Relational data into graph modelled data.

For this reason, several solutions have been proposed to support the translation of Relational data into RDF. Some of them focus on mapping the source schema into an ontology [38] and rely on a naive transformation technique in which every Relational attribute becomes an RDF predicate and every Relational values becomes an RDF literal. RDBToOnto [39] tool is used to derive ontologies accurately by using knowledge of database schema and data. This tool also identifies the taxonomies hidden in the data and use the knowledge discovered to derive ontologies.

There are some other approaches like R_2O [40] and D2RQ [41] to convert Relational data to RDF. R_2O is XML-based declarative language, that allows description of complex mapping expressions between Relational and ontology elements.

As discussed in [42], there are many approaches for RDB to RDF conversion. But none of this approach fulfils all the requirements for RDB2RDF mappings. All these approaches focus on mapping of Relational data to Semantic Web. But the problem targeted in this thesis is more general where Relational data is needed to be converted to Graph Databases.

3.1 Problem Statement

With the growth in internet data is growing exponentially. This growing data needs to be stored in databases which can support scalability and store large amount of data efficiently. These days, many non-Relational Databases are available in the market. The popularity of these databases is increasing due to their highly scalable nature, ability to store semi-structured or schema-less data, etc. These databases also come in many flavours. Software developers and DBA have to choose among these many available databases to suit their data requirements. Many organizations are also shifting their data models from traditional Relational Databases to new NoSQL databases. This leads to requirement of data migration algorithms for shifting from one schema to another.

With the advent of cloud based services many companies especially small companies rely on cloud services where you pay as-per-need basis. Most of the cloud platforms support non-Relational Databases for scalability issues [43]. Thus, there grows the need for data migration from Relational to cloud databases.

Data migration and integration have always been one of the most researched area. But with increasing variety of data storing formats, data migration has gained more interest. There can be many reasons for migrating data from one database to another. Recently available NoSQL databases have their own pros and cons. There can be reasons such as dissatisfaction with current database schema, change in requirements or increasing data for shifting from one database to another.

3.2. Research Gaps

Many algorithms has been proposed for data migration and schema-mapping between Relational to non-Relational and between non-Relational to non-Relational. While migrating from Relational to non-Relational the biggest challenge is to transfer schema information to schema-less databases. For schema transformations where Relational Database is the source database, three types of relations are considered -

one-to-one, many-to-one or one-to-many and many-to-many. Relational Databases have these relations managed by foreign keys. For Graph Databases which are mainly used for traversal, schema paths are generated from foreign key relations in Relational Database. Column-families in case of Column-oriented databases and nodes in case of Graph Databases are considered joinable according to their foreign key relation in Relational Databases. Many approaches have been suggested for database translation but there is no focus on data migration and query mapping between Relational and Graph Databases.

3.3. Objectives

In the light of above discussed research gaps following objectives have been formulated.

1. To design an algorithm to transform Relational Database to graph datbase and migrate the data from Relational to Graph Database without any information loss.
2. To design a methodology to convert SQL queries in Relational Database to cypher queries in grapg database.
3. To perform the comparative analysis of SQL and Cypher query language based on their execution time.

3.4. Research Methodology

Data migration from Relational Database to Graph Database is the focus here. Foreign key relations in Relational Database are used to transform data from Relational Database to Graph Database. Latest version of Neo4j is used as the Graph Database. A methodology for converting queries from SQL to cypher query language has also been proposed. Comparative analysis of SQL and CQL is performed on the basis of the execution time in milliseconds

Chapter 4 Transforming Relational Database to Graph Database

4.1 Preliminaries

(i) A Relational schema is denoted as $R_i(A_i)$ [44] where,
R_i → *i-th relation* and
A_i → *set of attributes of i-th relation*

(ii) The set of such Relational schema $R_1(A_1).....Rn(An)$ is called Relational Database schema.

(iii) The Relational Database R is set of tuples over all the Relational schema $R_1(A_1).....Rn(An)$.

(iv) A Graph Database is set of nodes connected via relationships. Graph Database G can be denoted as $G(n,r)$ where,
n → *set of nodes* and
r → *set of relationships*

Here n and r each have their own properties and represent some information. All the relationships r are labelled which define the type of relationship.

Following is the Relational Database R which will be used as the example database. Figure 4.1 shows the example Relational Database R. Attributes that belong to primary keys of that relation are underlined in the figure 4.1. A foreign key relation will be represented as $R_i.A \rightarrow R_j.B$, where there is a foreign key relation between the attribute A of relation R_i and attribute B of relation R_j. Relational Database R consists of set of tuples over 4 relations. These relations are explained in Table 4.1.

Table 4.1: Schema details of example Relational Database R

Relation Name	Attributes	Primary Key	Foreign Key Relation
User	UserId, Name	UserId	None
Pages	PageId, Title, Creator	PageId	Pages(Creator) $\xrightarrow{fk}$ User(UserId)
Comments	Cid, Text, Page, User	Cid	Comments(Page) $\xrightarrow{fk}$ Pages(PageId), Comments(User) $\xrightarrow{fk}$ User(UserId)
Follower	UserId, PageId	UserId, PageId	Follower(UserId) $\xrightarrow{fk}$ User(UserId), Follower(PageId) $\xrightarrow{fk}$ Pages(PageId)

User

UserId	Name
u101	John
u102	Mark
u103	Claire

Comments

Cid	Text	Page	User
C01	New era!	P201	u103
C02	Not SQL!	P201	u102
C03	Social Data	P202	u102

Pages

PageId	Title	Creator
P201	NoSQL	u101
P202	Graph databases	u102
P203	Network Security	u101
P204	Information systsem	u103

Follower

UserId	PageId
u101	P201
u101	P203
u102	P202
u102	P204
u103	P203
u103	P204

Figure 4.1: An example Relational Database R

4.2 Types of Relationships in Relational Database

The foreign key relations in Relational Database results in three types of relations – one-to-one, one-to-many or many-to-one and many-to-many [31]. For converting data from Relational to graph, it's important to study these relationships. These relationships show how the tuples of different relations are joinable. In graph *G* two nodes representing the tuples of two relations are connected only if those tuples are joinable. Two tuples t_1 and t_2 are said to be joinable if,

$$t_1 \; \varepsilon \; R_i \text{ and } t_2 \; \varepsilon \; R_j \; ,$$
$$R_i(A) \xrightarrow{fk} R_j(B) \text{ and}$$
$$t_1.A = t_2.B$$

In **one-to-one relationship**, each tuple belonging to one relation R_i has only one one joinable tuple in R_j and vice-versa. Whereas, in **one-to-many** or **many-to-one** relations each tuple in R_i has zero or many joinable tuples in R_j or vice-versa. While transforming from Relational to Graph Database, the representation for 1:1 and 1:N or N:1 relationships is similar in graph *G*.

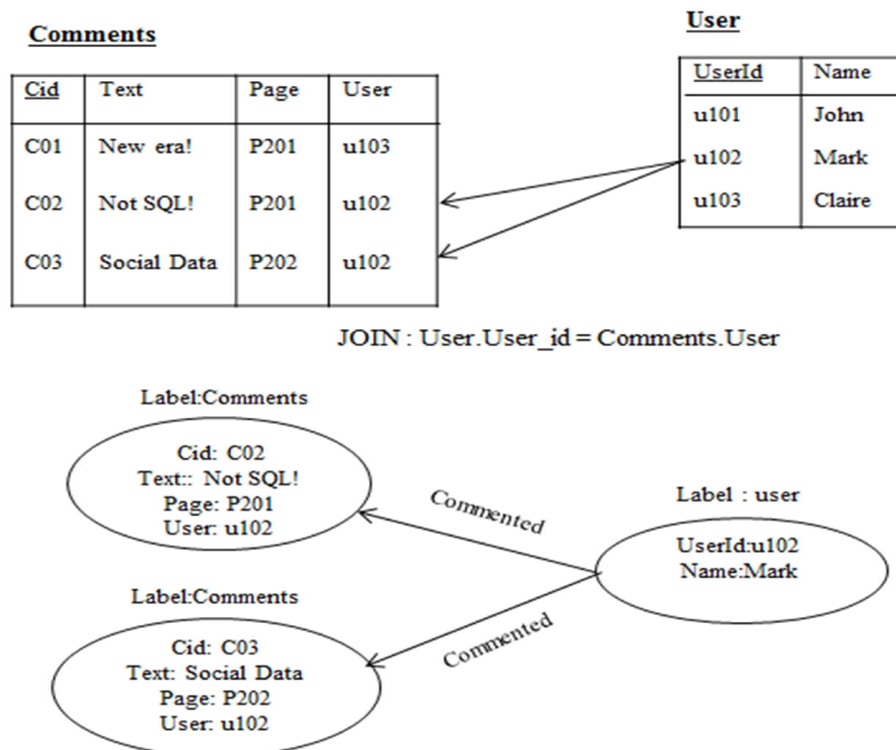


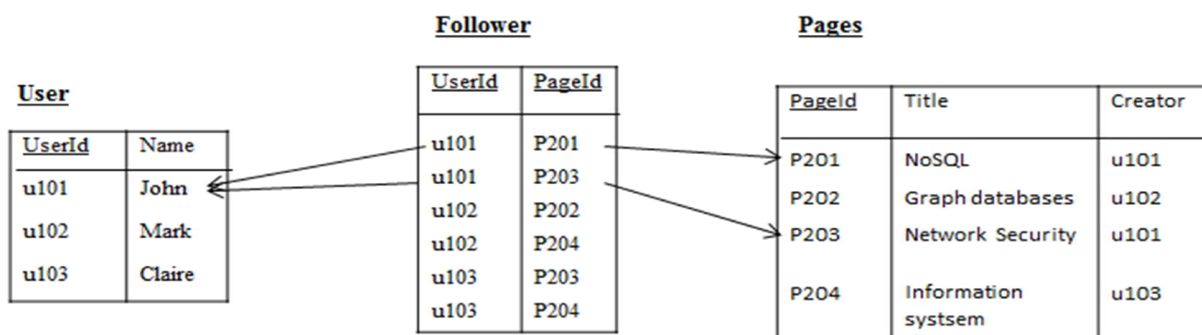
Figure 4.2: Transformation of 1:N Relationship from Relational to Graph Database

Figure 4.2 shows 1:N relation where each tuple on User relation can have zero or more joinable tuples in Comments relation. It also shows how these tuples are represented in graph *G*. It also shows the join condition of the two relations involved.

In **many-to-many relationships**, one tuple in R_i can have one or many matching tuples in R_j and one tuple in R_j can have many matching tuples in R_i. As shown in Figure 4.1, one User can follow many pages and one Page can be followed many Users. Such relationships are implemented by one intermediate relation which defines how two relations in M:N relationships are connected. These intermediate relations are called n2n relation.

N2N Relation: A relation is n2n if all the key attributes of R_i $(A_1,A_2,\ldots,A_n)$ have a foreign key relation. While converting such relations to graph *G*, an edge is added to graph *G* corresponding to each tuple of n2n relation and connecting the nodes represented by the relations that are connected via n2n relation.

Figure 4.3 shows, how Follower relation acts as n2n relation between User relation and Page relation.



User

UserId	Name
u101	John
u102	Mark
u103	Claire

Follower

UserId	PageId
u101	P201
u101	P203
u102	P202
u102	P204
u103	P203
u103	P204

Pages

PageId	Title	Creator
P201	NoSQL	u101
P202	Graph databases	u102
P203	Network Security	u101
P204	Information systsem	u103

JOIN: Follower.UserId = User.UserId AND Follower.PageId=Page.PageId

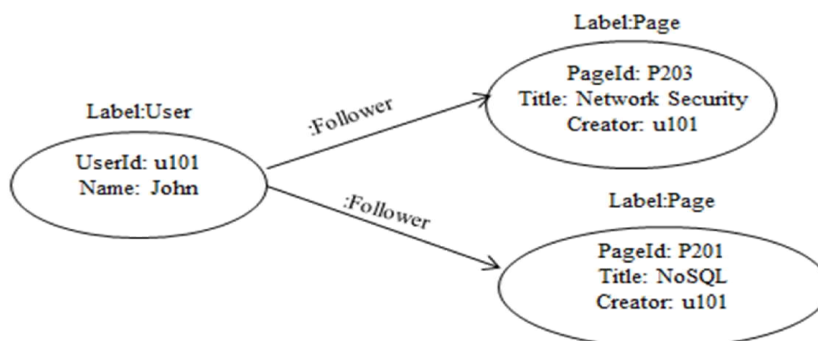


Figure 4.3: Transformation of N:M Relationship from Relational to Graph Database

4.3 Converting data from Relational to Graph Database

In this section, the algorithm that designs a Graph Database from the Relational Database has been proposed and discussed. Example Relational Database R shown in Figure 4.1 is used as input to the algorithm. The output Graph Database G is shown in Figure 4.5.

Algorithm 1 : *Transform Relational Database R to Graph Database G*

Input: Relational Database R

Output: Graph Database G

1. traversed[] ← Ǿ
2. allTables[] ← List of all tables in R
3. fkTables[] ← List of tables that are not n2n tables
4. n2nTable[] ← List of n2n tables in R
5. for each fkTable ∈ fkTables[]
    a. If fkTable.getExportedKeys() ≠ Ǿ
        i. Call traversefkTables();
6. traversed[] ← table
7. for each table ∈ fkTables[] and table ∉ traversed[]
    a. Call traverseRemaining();
8. traversed[] ← table
9. for each table ∈ n2nTables[]
    a. Call traverseN2NTables();
10. traversed[] ← table
11. Call createIndexes();

In the above algorithm initially, schema information of Relational Database R will be collected in variables described as follows,

- *allTables[]* → List of all tables. It conatins *{User,Page,Commnets,Folower}*

- *fkTables[]* → List of tables that are not n2n . It conatins *{User,Page,Comments}*
- *n2nTables[]* → List of n2n tables. It conatins *{Follower}*
- *Traversed[]* → Table name is added to this list after that table is traversed. Initially this list is empty.

Various modules called from Algorithm are briefly explained as follows:

- *traversefkTables()* → This module is called for each table in *fkTables[]* list that has atleast one exported key.
- *traverseRemaining()* → This module is called for all the tables in *fkTables[]* list that have not been traversed.
- *traverseN2NTables()* → This module is called for each n2n table.
- *createIndexes()* → This module create indexes.

Figure 4.4 shows the flowchart for conversion of Relational Database R to Graph Database G. It shows the flow of Algorithm defined above.

Relational Database R is given as input.

There are three *for loops*. Each loop calls a specific module until the condition holds. All three modules are called in a specific order.

In the end *createIndexes()* module is called to create indexes on all the nodes formed so far.

After processing through all the modules the output generated is the Graph Database G.

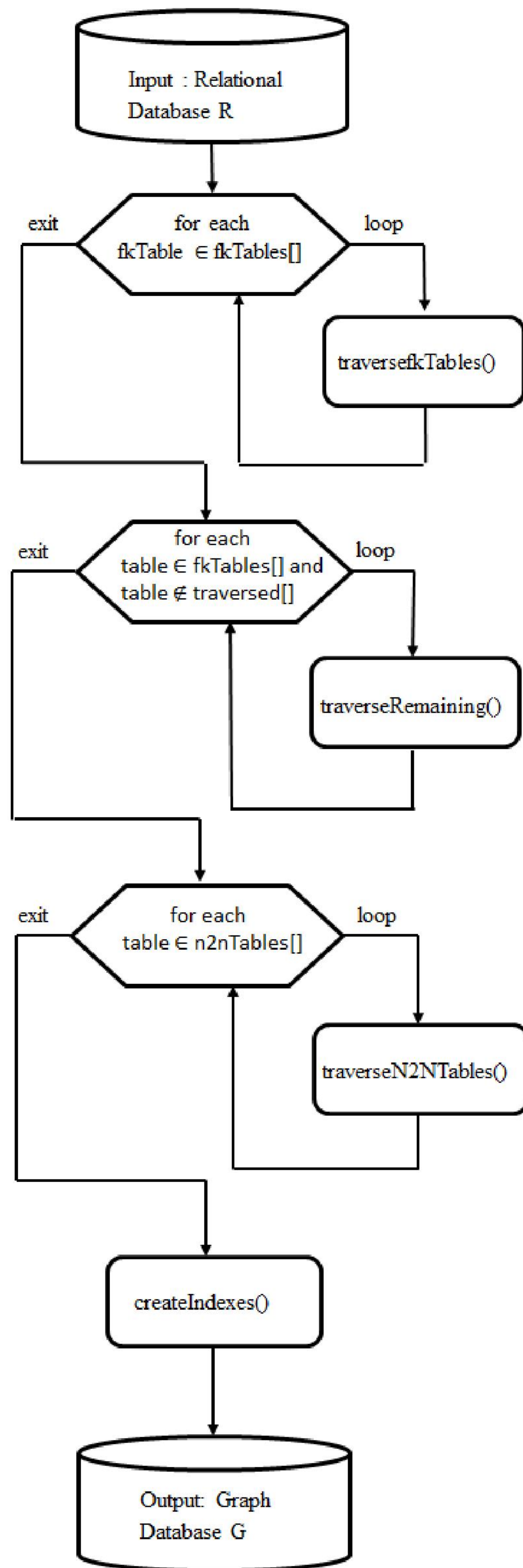


Figure 4.4: Flowchart

Different modules are called from the main module. Functionality and algorithm of all these modules is explained below.

Algorithm 2 : *traversefkTables()*

```
1.  exportTables[] ← fkTable.getExportedKey();
2.  For each tuple {t} ∈ fkTable
    a.  If table ∉ traversed[]
        i.  firstNode ← createNode();
        ii. addProperties() ← t.getColumnValues();
        iii. addLabel() ← fkTable.getName();
    b.  else
        i.  firstNode ← findNode(), where
            firstNode.label = table.getName() and property = t.PrimaryKey
    c.  For each exportable ∈ exportTables[] - n2nTables[]
        i.  For each tuple {t1} ∈ exportTable, where
            t1.foreignKey.Value=t.primaryKey.Value
            1.  If exportedTable ∉ traversed, than
                a.  secondNode ← createNode();
                b.  addProperties() ← t1.getColumnValues();
                c.  addLabel() ← exportTable.getName();
            2.  else
                a.  secondNode ← findNode(), where
                    secondNode.label= exportTable.getName()
                    and property = t1.PrimaryKey
            3.  addEdge() ← firestNode,secondNode
        ii. end for
    d.  traversed[] ← exportTable
3.  end for
```

f: traversefkTables() is called for each table in *fkTables[]* list that has atleast one exported key. Suppose *traversefkTables()* is called for *User* table. *exportTables[]* list will contain the list of tables that are using the fkTable's primary key as foreign key.

Eg., *page.creator* → *User.user_id* in *R*. So for *User* table *exportTables[]* contains *{Page,Comments,Follower}-{Follower}*. For each *fkTable* if that table is not traversed than its tuples are read. For each tuple *{t}* of *fkTable* a node *firstNode* is created in graph *G*, where *firstNode* has properties same as columns of tuple *t*. A label is also added to the node as the name of the *fkTable*. Such as for tuple {u101,John} a node is created with *User_id as u101 and name as John and label as User*. If the *fkTable* is already traversed than the node *firstNode* is found from Graph Database *G* with its property values same as the tuple *t*'s column's values. Now, for each *exportTable* in *exportTables[]* list, all the joinable tuples are read which have their foreign key attribute value same as the primary key attribute of tuple *t* of *fkTable*. Such as for table *User* the primary key value of the current tuple is *User_id = u101* and in table *Page* the joinable tuples for *User_id = u101* are *{P201,NoSQL,u101}* and *{P203,Network Security,u101}*. If this *exportTable* is not yet traversed than nodes are created in *G*, otherwise nodes are found in *G*, with values corresponding to all the joinable tuple's columns. Than for each such node corresponding to the joinable tuples found in *exportTable*, an edge is created in *G* between these nodes and *firstNode* created earlier. At the end of this *f: traversefkTables()* tables {*User, Comments, Page}* will be marked as traversed.

Algorithm 3 : *traverseRemaining()*
1. for each tuple {t} ∈ table a. Node ← createNode(); b. Node.addProperties() ← t.getColumnValues(); c. Node.addLabel() ← table.getName(); 2. end for

f: traverseRemaining() is called for all the tables in *fkTables[]* list that have not been traversed. For each such table a node is created in *G* corresponding to all the tuples of such table. Node's properties are same as tuple's columns and label is same as the table name in which the tuple is contained. In the end all such tables are marked traversed.

Algorithm 4 : *traverseN2NTables()*

1. importedTables[] ← table.getImportedKeyInfo()
2. for each tuple {t1} ∈ table
   a. for each importedTbale ∈ importedTables[]
      i. tuples ← findJoinableTuples(), where importedTbale.primaryKey.value = t1.foreignKey.value
      ii. nodes[] ← findNode() ∈ G, where node.label = importable and node.property = tuples.columns
   b. end for
   c. For all pairs of nodes ∈ nodes[]
      i. addEdge();
   d. end for
3. end for

f: traverseN2NTables() is called for each n2n table. In our example database *Follower* is a n2n table. For each table its imported key information is collected. Imported key information contains the table name of those tables whose primary keys are being used in the foreign key relation for each key attribute of n2n table. In our case, *User* and *Page* table are the tables whose primary keys *User_id* and *Page_id* respectively are used in *Follower* table in foreign key relation.

Follower.User_id → *User.user_id* and *Follower.page_id* → *Page_page_id.* Now for each tuple of n2n table *(Follower)* its joinable tuples are found in all the imported tables. For table *Follower*, tuple *{u101, P201}* its joinable tuples in User and Page table are {u101,John} and *{P201,NoSQL,u101}* respectively. Now for each joinable tuple find its corresponding node in G and create an edge between the nodes.

Algorithm 5 : *createIndexes()*
1. for each table ∈ fkTables[]   a. Create index such that label = table.name and index = table.primaryKey   2. end for

In the end, *indexes* are created for all node labels. The module defined above create indexes on the primary key attributes.

Properties that are most commonly used in the search criteria for querying the data are the ideal choice for making indexes. Indexes help in fast traversals, thus they are created on the properties that are used mostly in the search criteria. DBA involvement is required for the efficient choice of indexes.

Figure 4.5 shows the Graph Database G which is obtained from the Relational Database R shown in Figure 4.1 after the execution above proposed algorithm to convert Relational to Graph Database.

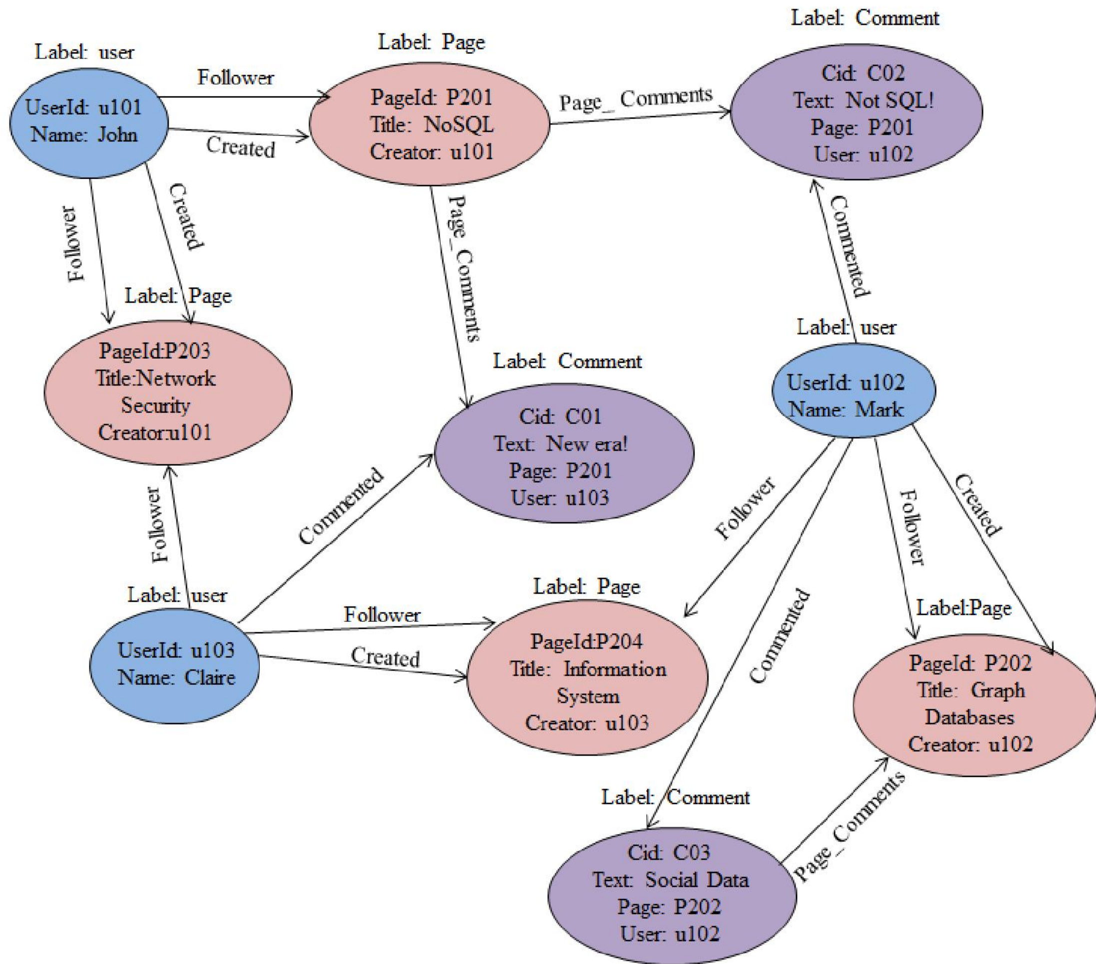


Figure 4.5: Graph Database G

Chapter 5 Implementation and Results

5.1. Implementation Environment

In this section, the algorithm proposed in previous section is used to transform IMDb movie database to Graph Database. SQL queries will be transformed into Cypher query language to traverse the graph data. A methodology is proposed to do query translation. Both SQL and Cypher queries will be run on their respective datasets and their execution time will be compared.

This algorithm is developed in Java 1.7 using Eclipse. The specifications of the machine which is used for implementation purpose are - Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz with 12 GB RAM and 64-bit OS.

5.2. Database Design

IMDb movie database is open source database. IMDb's data is available on internet in the form of plain text files. After downloading these files they are used to create a Relational Database using IMDbPY 4.1 [45]. Using a third-party tool eliminates any bias in the creation of the schema, which can lead to significant impact on the effectiveness and performance of the database. The initial database contained 20 relations with more than 44 million tuples. For the implementation purpose, only a subset of this whole database is used. The subset used has data over 6 relations.

Table 5.1 shows the schema of IMDb subset database that has been used for experiment. It shows the number of tuples over 6 relations.

Figure 5.1 sows the ER diagram of the IMDb subset database used for experiment.

Table 5.1: IMDb Database subset used for implementation (Primary Key, *Foreign Key*)

Relation	Attributes	Primary keys	Number of tuples
Movie	**Id**, Title, year	Id	80,853
Person	**Id**, name	Id	126,517
Character	**Id**, name	Id	93,476
Role	**Id**, type	Id	11
Cast	***movieId, personId, characterId, roleId***	movieId, personId, characterId, roleId	396,347
MovieInfo	**Id**, *movieId*, info	Id	89,339

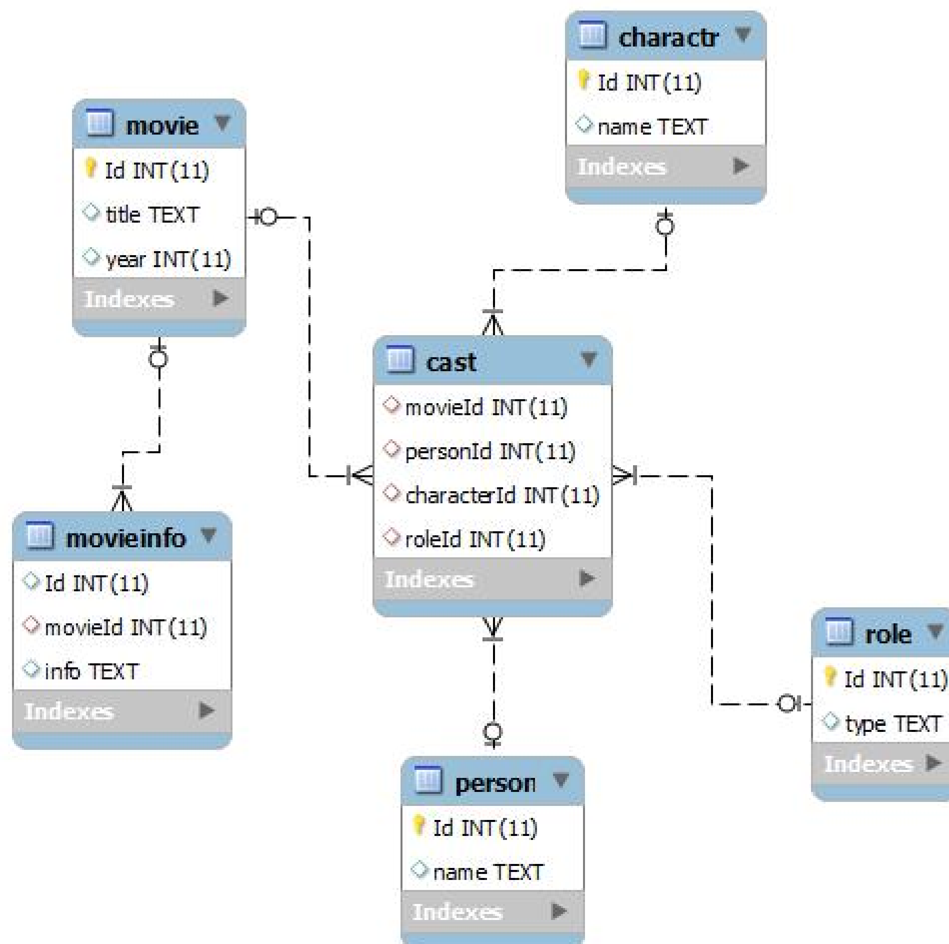


Figure 5.1 : ER Diagram of IMDb subset Database

5.3. Experimental Results

After running the Algorithm 1 proposed in Chapter 4 on the IMDb database explained in Table 5.1, a Graph Database is created. The resulting Graph Database consists of 80,853 nodes with label "Movie", 126,517 nodes with label "Person", 93,476 nodes with label "Character", 11 nodes with label "Role" and 89,339 nodes with label "MovieInfo". Tuples belonging to the cast relation convert to form 396,347 edges between "Movie" and "Person" , 396,347 edges between "Movie" and "Character", 396,347 edges between "Movie" and "Role", 396,347 edges between "Person" and "Character", 396,347 edges between "Person" and "Role" and 396,347 edges between "Character" and "Role" with relationship name as "Cast". These "Cast" edges connect Movie, Person, Character, Role nodes. There are also 89,339 edges between Movie and MovieInfo nodes with relationship name as "MovieInfo".

Following are the five snapshots shown in Figure 5.2, 5.3, 5.4, 5.5 and 5.6 shows the number of nodes, Figure 5.7 and 5.8 shows the number of edges formed for each label after executing the Algorithm 1 of transforming Relational Database to Graph Database. The algorithm is run on the Relational Database explained in Table 5.1 and Figure 5.1.

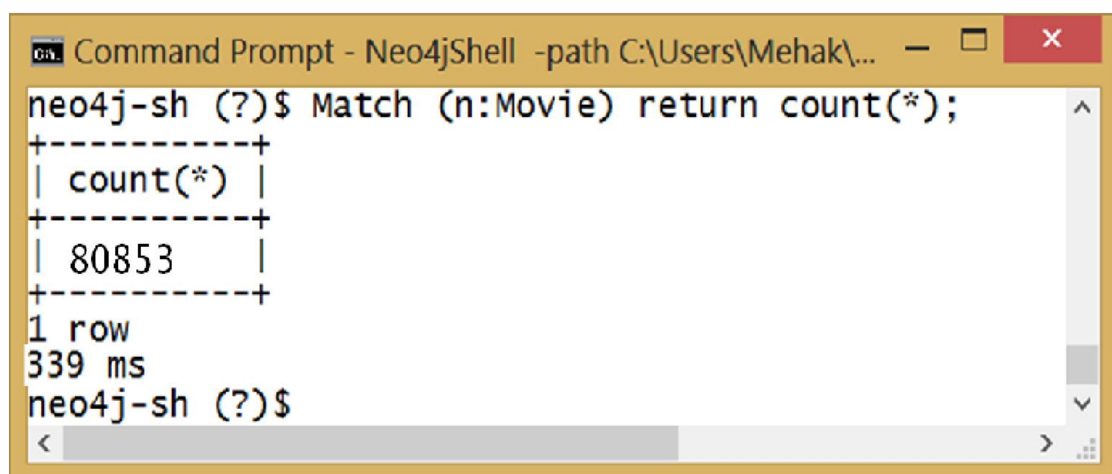


Figure 5.2: Number of nodes of label "Movie"

```
Command Prompt - Neo4jShell -path C:\Users\Mehak\...
neo4j-sh (?)$ Match (n:Person) return count(*);
+----------+
| count(*) |
+----------+
| 126517   |
+----------+
1 row
578 ms
```

Figure 5.3: Number of nodes of label "Person"

```
Command Prompt - Neo4jShell -path C:\Users\Mehak\...
neo4j-sh (?)$ Match (n:Character) return count(*);
+----------+
| count(*) |
+----------+
| 93476    |
+----------+
1 row
454 ms
```

Figure 5.4: Number of nodes of label "Character"

```
Command Prompt - Neo4jShell -path C:\Users\Mehak\...
neo4j-sh (?)$ Match (n:MovieInfo) return count(*);
+----------+
| count(*) |
+----------+
| 89339    |
+----------+
1 row
341 ms
```

Figure 5.5: Number of nodes of label "MovieInfo"

Command Prompt - Neo4jShell -path C:\Users\Mehak\...

```
neo4j-sh (?)$ Match (n:Role) return count(*);
+-----------+
| count(*)  |
+-----------+
| 11        |
+-----------+
1 row
0 ms
```

Figure 5.6: Number of nodes of label "Role"

Command Prompt - Neo4jShell -path C:\Users\...

```
neo4j-sh (?)$ Match (r:Cast) return count(*);
+-----------+
| count(*)  |
+-----------+
| 2378082   |
+-----------+
1 row
9210320 ms
```

Figure 5.7: Number of edges with name "Cast"

Command Prompt - Neo4jShell -path C:\Users\Mehak...

```
neo4j-sh (?)$ Match (r:MovieInfo) return count(*);
+-----------+
| count(*)  |
+-----------+
| 89339     |
+-----------+
1 row
93456 ms
```

Figure 5.8: Number of edges with name "MovieInfo"

5.4. Query Translation Methodology

Query languages have always been the key towards success of a database. Graph query languages mainly focus on traversal paths [46]. In this section, a methodology is proposed to translate queries from SQL to Cypher query language. Cypher query language (CQL) is one of the most used graph query language. The methodology to translate queries from SQL to Cypher is explained below with the help of an example SQL query. This SQL query will be translated step by step to fully convert it to Cypher query language. Five steps are explained below where each SQL clause is mapped to clauses supported by cypher query language (CQL).

Example SQL Query:

*SELECT * FROM Movie M, Person P, Cast C WHERE M.Id=C.movieId and P.id=C.personId and M.Id="xxxxxx";*

1) FROM clause → MATCH clause:

In SQL, *FROM* clause specifies the tables from which the data has to be extracted. These tables are involved in the joins in the where clause. In Cypher query, *MATCH* clause is used define the pattern that needs to be traversed. So, the table names used in *FROM* clause are used as the label names of nodes in the *MATCH* clause. So w.r.t example SQL query following is added to the *MATCH* clause of CQL

(n:movie), (m:person), (p:cast)

2) WHERE clause → MATCH clause:

In SQL, *WHERE* clause specifies how tables are joined. In Cypher, this information can be used to write *MATCH* clause which defines the node-relationship pattern that is to be matched in the graph. As in the *WHERE* clause of SQL query there are two joins between movie and cast and between person and cast. So, these three joins are translated o the following *MATCH* clause.

MATCH (n:movie)-[p:_cast]-(m:person)

3) WHERE clause → WHERE clause:

WHERE clause is filtering clause in both SQL and CQL. Other than defining joins among various tables, WHERE clause is also specifies some filtering values such as

M.Id given in the above SQL query. Such information in WHERE clause of SQL is directly translated to WHERE clause of CQL. Hence, WHERE clause is written as below in CQL.

WHERE n.Id= "xxxxxx"

4) SELECT clause → RETURN clause:

SELECT clause tells about which attributes are required in the result of the SQL query. Same is achievd by the *RETURN* clause in CQL. As in the example query the *SELECT* clause has (*) so it means all attributes of the rows selected are required in the result. So, in CQL *RETURN* clause all nodes are returned with all the properties. So RETURN clause will simply contain node names without specifying any of their specific properties.

RETURN n, m, p;

5) Indexes :

The attributes whose values are used as input in equality comparison in *WHERE* clause of the SQL query can be used as indexes in CQL. With Neo4j 2.0 version CQL can use indexes. So, CQL's INDEX clause will become as:

USING INDEX n:movie(Id)

Now, after combining all the above clauses, the final CQL query will become as following:

MATCH (n:movie)-[p:_cast]-(m:person) USING INDEX n:movie(Id) where n.Id= "xxxxxx" return n, m, p;

5.5. Performance comparison of SQL and CQL

In this section, various queries will be translated from SQL to CQL and will be executed on their respective database. Their execution time will be compared, to know the difference between the performance of Relational and Graph Database.

Table 5.2 shows the details of the five queries which are used to do comparative analysis of Relational query language (SQL) and graph query language (CQL).

Table 5.2: Five queries with the relations involved in their joins

Query	Query Statement	Relations from which data is to be fetched
Query 1	Find the movie information from movie and movie_info for a given movie title	Movie, MovieInfo
Query 2	Find person, role and movie information for a given person name	Person, Role, Cast, Movie
Query 3	Find person information for a given character name of a movie	Person, Character, Cast, Movie
Query 4	Find all information about person, character and movie for a given role type	Person, Role, Movie, Cast
Query 5	Find all the information about a movie for a given movie title	Movie, Person, Character, Role, Cast, MovieInfo

Following are the five queries described in Table 5.2 with their syntax in both SQL and Cypher query language

Query 1 : **Find the movie information from movie and movie_info for a given movie title**

SQL	CQL
SELECT * from movie M, movie_info MI WHERE M.Title= "xxxxxx" AND M.Id = MI.Id	MATCH (n:movie)-[r:_movie_info]-(m:movie_info) USING INDEX n:movie(title) where n.title= "xxxxxx" return n,m;

Query 2 : **Find person, role and movie information for a given person name**

SQL	CQL
SELECT * from person P, role R, Movie M, cast C WHERE P.name = “xxxxxx” AND C.personId = P.Id AND C.roleId=R.Id AND C.movieId=M.Id	MATCH (n:movie)-[r:_cast]-(m:person)-[r1:_cast]-(p:role) USING INDEX m:person(name) where m.name= “xxxxxx” return n,m,p;

Query 3 : **Find person information for a given character name of a movie**

SQL	CQL
SELECT P.name from person P, character CH, cast C, movie M WHERE CH.name = “xxxxxx” AND M.title = “xxxxxx” AND C.characterId = CH.Id AND C.movieId = M.Id AND C.personId=P.Id	MATCH (n:movie)-[r:_cast]-(m:person)-[r1:_cast]-(p:character) USING INDEX m:person(name) where p.name= “xxxxxx” and n.title= “xxxxxx” return m.name;

Query 4 : **Find all information about person, character and movie for a given role type**

SQL	CQL
SELECT P.name, CH.name, M.title from person P, role R, character CH, cast C, movie M WHERE R.type = “xxxxxx” AND C.roleId=R.Id AND C.characterId = CH.Id AND C.movieId = M.Id AND C.personId=P.Id	MATCH (n:movie)-[r:_cast]-(m:person)-[r1:_cast]-(p:character)-[r2:_cast]-(q:role) USING INDEX q:role(type) where q.type= “xxxxxx” return m.name, p.name, n.title;

***Query 5* : Find all the information about a movie for a given movie title**

SQL	CQL
SELECT * from person P, role R, character CH, cast C, movie M, movie_info MI WHERE M.title= "xxxxxx" AND C.movieId = M.Id AND C.roleId=R.Id AND C.characterId = CH.Id AND C.personId=P.Id AND MI.movieId=M.Id	MATCH (k:movie_info) -[r0:_movie_info]-(n:movie)-[r:_cast]-(m:person)-[r1:_cast]-(p:character)-[r2:_cast]-(q:role) USING INDEX n:movie(title) where n.title= "xxxxxx" return k,n,m,p,q;

Queries mentioned in the Table 5.2 are executed 10 times to compute the average execution time. Figure 5.9 shows the difference between execution time of SQL and CQL in milliseconds for the above five queries. It shows the average execution time for 10 consecutive executions.

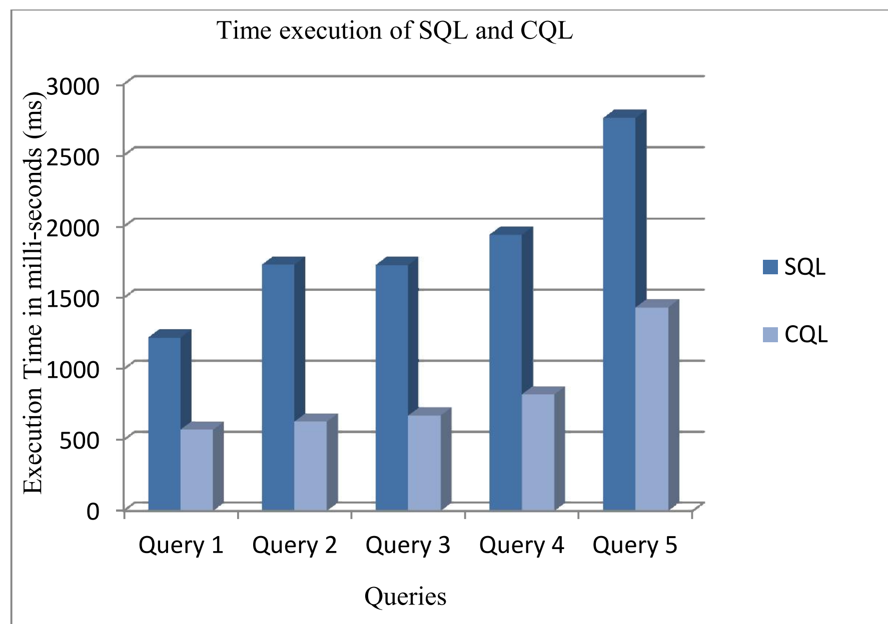


Figure 5.9: Comparison of SQL and CQL on the basis of execution time

Figure 5.9 represents the execution time of five queries described in Table 5.2 in both SQL and CQL in milliseconds along the vertical axis. Queries described in Table 5.2 are represented along the horizontal axis. The bar graph plotted in figure 5.9, shows that Cypher queries are much faster as compared to the SQL queries. Joins in SQL queries are slower than the pattern matching in the Cypher queries.

Chapter 6 Conclusion and Future Scope

6.1. Conclusion

In this thesis an algorithm has been suggested for automatically converting Relational data into Graph Database. This algorithm supports new features of Neo4j 2.0 such as labels and indexes in Graph Database. Labels and indexes helps in fast traversal of graph data. A methodology has also been proposed for converting SQL queries to Cypher query language. This methodology has been applied for translating queries from SQL to Cypher query language (CQL). In the last a comparison of SQL and CQL queries shows that graph query languages are much more efficient as compared to SQL. Joins used in Relational query languages are simply converted to a traversal path in graph query language. The comparative analysis of both SQL and CQL shows that joins are much slower as compared to traversals in graph query language.

6.2. Summary of Contributions

The contributions made by the research work presented in this thesis are summarized as follows:

- Survey of the existing methods available for converting Relational data to graph modelled data.
- Comparison of Relational Databases and Graph Databases on the basis of their features such as scalability, data models, transactional properties and query languages.
- Study of applications of Graph Databases in real-world commercial projects such as Ebay, SNAP, Glassdoor, etc.
- A novel approach for transforming data from Relational Database to Graph Database.
- Methodology to convert relation queries in SQL to graph queries in Cypher.
- Comparative analysis of SQL and CQL on the basis of their execution time.

6.3. Future Scope

The field of data migration can be further explored for transforming data from Relational to various other NoSQL databases. The work presented in this thesis can be further enhanced to transform data from one Relational Database to any other Graph Databases such as OrientDB, etc. also.

New methodologies can be designed using schema information of Relational Database to convert SQL queries to Cypher queries.

References

1. E.F. Codd, "A Relational Model of Data for Large Shared Data Banks", *Communications of the ACM*, vol. 13, no. 6, pp. 377-387.
2. K. Kaur and R. Rani, "Modeling and Querying Data in NoSQL Databases", *IEEE International Conference on Big Data*, pp. 7, 6-9, Oct. 2013, Silicon Valley, CA, 2013.
3. C. Strauch, "NoSQL Databases", *ACM New York, NY, USA*, pp. 2-31, 2010.
4. The memcached website. [Online]. Available: http://www.memcached.org/.
5. The Riak website. [Online]. Available: http://basho.com/riak/
6. Project Voldemort: A Distributed Database. [Online]. Available: http://www.project-voldemort.com/voldemort/
7. A. Lakshman and P. Malik, "Cassandra a Decentralized Structured Storage System", *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35-40, 2010.
8. R. Arora and R.R. Aggarwal, "An Algorithm for Transformation of Data from MySQL to NoSQL (MongoDB)", *International Journal of Advanced Studies in Computer Science and Engineering (IJASCSE),* vol. 2, no. 1, 2013.
9. R. Arora and R.R. Aggarwal, "Modeling and Querying Data in MongoDB", *International Journal of Scientific and Engineering Research (IJSER),* vol. 4, Issue 7, pp. 141-144, July 2013.
10. The MongoDB website. [Online]. Availble: http://www.mongodb.org/
11. P. Raichand, "Query Execution and Effect of Compression on NoSQL Column Oriented Data Store using Hadoop and HBase", *International Journal of Scientific and Engineering Research (IJSER) - (ISSN 2229-5518),* vol. 4, no. 9, 2013.
12. F. Changetal, "Bigtable: A Distributed Storage System for Structured Data," in *Proceedings of the 7th symposium on Operating systems design and implementation OSDI '06, Berkeley, CA*, pp. 205-218, 2006.
13. P. Raichand and R. Rani, "A Short Survey of Data Compression Techniques for Column Oriented Databases", *Journal of Global Research in Computer Science*, vol. 4, no. 7, pp. 43-46, July 2013.

14. The neo database. [Online]. Available: http://dist.Neo4j.org/ Neo-technology-introduction.pdf.
15. Neo4j. Home. [Online]. Available: http://Neo4j.org.
16. C. Berge, "The theory of Graphs and their Applications”, *Wiley*, 1962.
17. S. Berretti, A. D. Bimbo and E. Vicario, “Efficient Matching and Indexing of Graph Models in Content-based Retrieval” in *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 23, 2001.
18. J. Lee, J. Oh and S. Hwang, “STRG-Index: Spatio-temporal Region Graph Indexing for Large Video Databases”, in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 718-729, 2005.
19. I. Robinson, J. Webber and E. Eifrem, “Graph Databases”, *USA: O'Reilly,* pp. 8-31, 2013.
20. C. Vicknair, M. Macias and Z. Zhao, “A Comparison of a Graph Database and a Relational Database: A data provenance perspective." in *Proceedings of the 48th annual Southeast Regional Conference, ACM*, pp. 42, 2010.
21. S. Gilbert and N. Lynch, “Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”, *ACM SIGACT News*, vol. 33, no. 2, pp. 51-59, 2002.
22. J.J. Miller, “Graph Database Applications and Concepts with Neo4j”, in *Proceedings of the 2013 Southern Association for Information Systems*, 2013.
23. 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.
24. M.A. Rodriguez and P. Neubauer, “Constructions from Dots and Lines”, *Bulletin of the American Society for Information Science and Technology, American Society for Information Science and Technology*, vol. 36, no. 6, pp. 35-41, August 2010.
25. Neotechnology. [Online]. Available: http://www.neotechnology.com/ebay-walmart-adopt-neo4j-graph-transforming-retail/.
26. Neo4j Blog. [Online]. Available: http://blog.Neo4j.org/2013/12/Neo4j-20-ga-graphs-for-everyone.
27. F. Holzschuher and R. Peinl, “Performance of Graph Query Languages - Comparison of Cypher, Gremlin and Native Access in Neo4j”, in *Proceedings of the Joint EDBT/ICDT Workshops, ACM,* pp. 195-204, 2013.

28. M. Buerli and C.P.S.L. Obispo, "The Current State of Graph Databases", *Department of Computer Science, Cal Poly San Luis Obispo, mbuerli@ calpoly. edu.*, 2012.
29. R. Angles, "A Comparison of Current Graph Database Models", in *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops (ICDEW), IEEE 28th International Conference*, pp. 171-177, 2012.
30. R. Angles and C. Guti´errez, "Survey of Graph Database Models", *ACM Computing Surveys,* vol. 40, no. 1, 2008.
31. C. Li, "Transforming Relational Database into HBase: A Case Study", *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference*, July 2010.
32. P. Atzeni, P. Cappellari, R. Torlone, P.A. Bernstein and G. Gianforme, "Model-independent Schema Translation", *The VLDB Journal—The International Journal on Very Large Data Bases,* vol. 17, no. 6, pp. 1347-1370, 2008.
33. N. Shadbolt, T. Berners-Lee and W. Hall, "The Semantic Web Revisited", *Intelligent Systems, IEEE*, vol. 21, no. 3, pp. 96–101, 2006.
34. K.C.C. Chang, B. He, C. Li, M. Patel and Z. Zhang, "Structured Databases on the Web: Observations and Implications", *ACM SIGMOD Record*, vol. 33, no. 3, pp. 61-70, 2004.
35. A. Langegger, W. Wöß and M. Blöchl, "A Semantic Web Middleware for Virtual Data Integration on the Web", *In The Semantic Web: Research and Applications, Springer Berlin Heidelberg,* pp. 493-507, 2008.
36. L. Ma, X. Sun, F. Cao, C. Wang, X. Wang, N. Kanellos, D. Wolfson and Y. Pan, "Semantic Enhancement for Enterprise Data Management", *In The Semantic Web-ISWC, Springer Berlin Heidelberg*, pp. 876-892, 2009.
37. C. Patel, S. Khan, K. Gomadam and V. Garg, "TrialX: Using Semantic Technologies to Match Patients to Relevant Clinical Trials Based on Their Personal Health Records", *Web Semantics: Science, Services and Agents on the World Wide Web,* vol. 8, no. 4, pp. 342-347, 2010.
38. J. Sequeda, M. Arenas, and D.P. Miranker, "On Directly Mapping Relational Databases to RDF and OWL", in *Proceedings of the 21st International Conference on World Wide Web, ACM,* pp. 649-658, 2012.

39. F. Cerbah, “Learning Highly Structured Semantic Repositories from Relational Databases”, *In The Semantic Web: Research and Applications, Springer Berlin Heidelberg,* pp. 777-781, 2008.
40. J.B. Rodriguez and A. Gomez-Perez, “Upgrading Relational Legacy Data to the Semantic Web”, in *Proceedings of the 15th International Conference on World Wide Web, ACM,* pp. 1069-1070, 2006.
41. C. Bizer, “D2r Map - A Database to RDF Mapping Language”, 2003.
42. M. Hert, G. Reif and H.C. Gall, “A comparison of RDB-to-RDF Mapping Languages”, in *Proceedings of the 7th International Conference on Semantic Systems, ACM*, pp. 25-32, 2011.
43. J. Han, M. Song and J. Song, "A Novel Solution of Distributed Memory NoSQL Database for Cloud Computing", in *Proceedings Computer and Information Science (ICIS), IEEE/ACIS 10th International Conference*, pp. 351-355, 2011.
44. W. Hu and Y. Qu, “Discovering Simple Mappings between Relational Database Schemas and Ontologies”, *In The Semantic Web, Springer Berlin Heidelberg,* pp. 225-238, 2007.
45. J. Coffman and A.C. Weaver, “A Framework for Evaluating Database Keyword Search Strategies”, *in Proceedings of the 19th ACM International Conference on Information and Knowledge Management, ACM*, pp. 729-738, 2010.
46. P.T. Wood, “Query Languages for Graph Databases”, *ACM SIGMOD Record*, vol. 41, no. 1, pp. 50-60, 2012.

List of Publications

1. Mehak Gupta and Rinkle Rani Aggarwal, "Transforming Relational Database to Graph Database Using Neo4j", *in Proceedings of Elsevier, Second International Conference on "Emerging Research in Computing, Information, Communication and Applications" (ERCICA-14)*, NMIT Bangalore, August 01-02, 2014.
   [Accepted]