A Novel Framework for Querying Multiparadigm Databases

A Thesis submitted for
the award of degree of
DOCTOR OF PHILOSOPHY

by

Karamjit Kaur
(951103002)

Under the guidance of

Dr. Rinkle Rani
Assistant Professor
Computer Science and Engineering Department
Thapar University, Patiala

Computer Science and Engineering Department
Thapar University, Patiala – 147004, INDIA

July 2015

Contents

Scholarship Awarded

List of Figures

List of Tables

Dedicated to

my loving parents and amazing husband

Certificate

I hereby certify that the work which is being presented in this thesis titled **"A Novel Framework for Querying Multiparadigm Databases"**, for the award of degree of **"Doctor of Philosphy"** 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. Rinkle Rani and refers other researchers works which are duly listed in the reference sction.

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

Kcheema

(Karamjit)
951103002

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

Rinkle 23/10/15

(Rinkle Rani)
Assistant Professor
Computer Science and Engineering Department
Thapar University
Patiala

Acknowledgment

First of all, I express my gratitude to the Almighty, Who blessed me with the zeal and enthusiasm to complete this work successfully.

I thank my supervisor, Dr. Rinkle Rani, Assistant Professor, Thapar University, Patiala for her suggestions and constant support during this research. I am grateful to her for motivating and inspiring me to go deeply into the field of mutiparadigm databases and supporting me throughout the life cycle of PhD studies.

I am also thankful to Dr. Deepak Garg, Head of CSED, for his guidance through the early years of chaos and confusion. I am thankful to my Doctoral committee members for their constructive comments and regularly ensuring the progress of my research work. My deep regards to Dr. Prakash Gopalan, Director, Thapar University for giving me access to facilities, which have been immensely helpful for the completion of my work.

I wish to thank the faculty and staff members of CSED of Thapar University, Patiala for their co-operation and support. I am also very thankful to my friends of CSED research lab for their continuous motivation and moral support.

My parents provided an environment that guided the first steps of the journey and encouraged the later ones. My brother and sister-in-law helped me in every possible way at each and every step during last few years. My in-laws family provided me an excellent conducive environment for continuing my research. Especially my grand-father-in-law and mother-in-law were highly inquisitive about the progress of my research work. Support of my mother-in-law played a very crucial role in my PhD work. My husband has been my pillar of strength throughout. He has played many roles — technical mentor, best-friend and foremost a continuous source of motivation.

Karamjit

List of Abbreviations

HIS	Hospital Information System
EMR	Electronic Medical Record
API	Application Programming Interface
SQL	Structured Query Language
CAP	Consistency Availability and Partition Tolerance
GAV	Global-As-View
LAV	Local-As-View
JSON	Java Script Object Notation
KBoD	Knowledge-Base of Data
ACID	Atomicity Consistency Isolation Durability
BASE	Basically Available Soft-state Eventually Consistent
CRUD	Create Read Update Delete
NoSQL	Not Only SQL
PolyglotHIS	Polyglot-persistent Healthcare Information System

Abstract

Relational databases evolved in accordance to the prevailing technological requirements and constraints that had suitability, applicability and relevance at that time. However, times have changed and also the contemporary requirements. To alleviate the problems associated with relational databases in handling present big data, which is predominantly un-structured, a new class of databases has emerged, known as NoSQL databases. As and when a new class of data-storage emerges and becomes popular, researchers start working towards its integration with the existing databases. Likewise, with the widespread use of NoSQL data-stores, the problem of integrating them with existing database technology has became a challenge. The goal is to select the most appropriate data storage technology that meets the specific requirements of each module of the application.

Amalgamation of different databases within an application is known as *Multiparadigm approach* or *Polyglot-persistence*. Persistence needs of applications are progressing from mostly relational to a mixture of data-stores. For example various modules of Health-care Information System (HIS) use different data-stores to model data closer to their semantic usage. The researcher has showcased the applicability of our multipradigm framework in HIS, considering the variety of data and diverse categories of NoSQL data-stores with which they may be managed. But the concept is equally well applicable to any other application area, where different parts of the application deals with distinct data formats.

The researcher has implemented a healthcare information system – *PolyglotHIS*, which makes use of one relational and two NoSQL data-stores. This coalition of data-stores is not arbitrary, instead it is prudently chosen on the basis of careful analysis of alternative data-stores. Each involved data-store has its own specific advantage. Relational data-stores are preferred for data pertaining to financial transactions, since they support transactional properties. Employees' payroll data, patients' billing

information and financial component of pharmacy department are handled by the relational database – *PostgreSQL*. NoSQL data-stores supports *BASE* (Basically Available Soft-state, Eventually-consistent) properties, which is the opposite of *ACID* and therefore not suitable for transactions.

Other two NoSQL data-stores used in PolyglotHIS are: MongoDB and Neo4j. *MongoDB*, the most widely-used document-database today, is *schema-less* and best suited for storing unstructured or semi-structured data, such as laboratory reports, laboratory images, instrument manuals, photos of the patients and doctors, etc. The data containing in-built relationships, such as blood relation between various patients thereby helping doctors to trace presence of any hereditary disease(s) by examining these relationships, are stored in graph database (i.e. Neo4j). Inter-linking between symptoms has also been stored in Neo4j graph database to assist the doctor in visualizing the links between symptoms and the disease, leading to quicker diagnosis.

Integration of multiple data-stores is facilitated through usage of *multiple co-operative agents*, which make up mediation layer of the system. Provision of a framework to represent knowledge about schemas of constituent data-stores in a unified representation scheme is achieved with the help of *Datalog* facts and rules. Datalog, which is a declarative logic programming language, is used to store sets of facts and rules, helps in the storing and inferring about the capabilities of data-stores used in the PolyglotHIS. Homogenization of results obtained from heterogeneous data-stores has been made possible due to the support for *JSON format* within all the involved data-stores.

This proposed approach is novel in the way various data-stores are integrated, making use of the NoSQL data-stores, which represent the modern data storage technology. Apart from NoSQL technologies, multiple co-operative agents are used. In terms of performance, the latency caused due to presence of the mediation layer in PolyglotHIS is negligible and becomes totally insignificant as the dataset size increases. Undoubtedly, the overall complexity of the system increases as there is an impedance mismatch between various data-stores in terms of data modeling and query languages; however, proposed solution is still advantageous because of

its ability to store the data according to its usage, thereby simplifying the overall programming model. Decentralized data processing is also made possible due to the use of multiple data-stores.

Chapter 1

Introduction

This chapter provides an overview of the thesis along with purpose and significance of the research work. The need for the research work done is briefed initially, followed by the importance of the research. Various research questions or gaps in existing literature that helped us to formulate our problem statement are also presented in this chapter.

Based on the gaps identified in the literature, objectives of the thesis were formed. The research methodology followed to achieve the formulated objectives are also described. Contributions and organization of the thesis are also presented in the chapter. This chapter summarizes the complete thesis enabling the reader to quickly understand the purpose, significance, objectives and the contribution of the thesis in the area of integrating data from multiple heterogeneous data-sources.

This thesis proposed and justified the use of polyglot-persistence instead of traditional one data-model based data storage. Polyglot-persistent softwares make use of more than one type of data-store within an application based upon the storage and query requirements of the application.

1.1 Research Motivation

Till now, relational database was the de-facto database. By default, a database used to mean data stored in tables. Relational model is suitable for many kinds of data, particularly when there is a requirement to pick apart data and re-assemble it in diverse ways for different purposes. NoSQL data stores are getting more popular as they handle the problem of scalability very well. But these data stores are still new in the field and will take time to mature, to get standardized and become robust. Besides, in order to run effectively on a cluster, most NoSQL databases have limited transactional capability. With the advent of these variety of non-relational storage solutions, programmers have many options to select from. Within one application different classes of NoSQL databases can be used simultaneously. This concept is known as polyglot persistence [1].

Within a single application different modules deal with different types of data sets, for example some modules may have to interact with graphs, other modules access data stored as documents etc. A single database system cannot efficiently handle all these types of data-sets. If instead of one particular database, a hybrid approach is available then a programmer can choose multiple databases for different data sets. For example, for storing user sessions Redis can be used, since it provides rapid access for reads and writes and this session information does not need to be durable. Similarly, for transactional updates, relational databases are preferred; for shopping cart implementation, Riak will be preferred as it provides high availability across multiple locations and can merge inconsistent writes; for storing recommendations, Neo4j will perform the best as it a graph database.

As explained above, each database has different strengths and weaknesses, which need to be aligned for particular application requirements. There are few available tools such as Clustrix, ScaleDB, NimbusDB, VoltDB etc. which use only a few classes of NoSQL data stores, but not all classes of databases. Thus, there is a need for a user friendly interface to query different types of data from the hybrid database. Further, the implementation details of the system should be hidden from the user. Various features which determine the selection of a particular data store

are – throughput requirements of reads and writes, whether the application needs to handle data distributed across nodes and serve query requests from users even when some nodes fail, whether the application needs offline data processing capabilities, application's availability requirements, application's data consistency requirements etc.

1.2 Significance of the Research

We argue for a system based on the premise that instead of forcing all type of data to fit in row-column format of relational databases, multiple data-stores should be used; each storing and representing data closer to its actual representation and usage. We have demonstrated the applicability of our proposed system in the domain of healthcare information systems (HIS). Different departments in hospital deal with diverse types of data and hence should deploy distinct data-stores. Hospitals are extremely complex institutions where large departments and units coordinate to provide the needed care for patients.

Naive approach for implementation of HIS is to use any one popular database to store and process all the data. Nonetheless, any single data-model cannot efficiently store and process multitude of data generated by healthcare institutions. For example, not all data fit well into row-column format of traditional relational databases. However, modern NoSQL data-stores allow data to be stored in a form closer to their actual representation and usage. There is an increasing dependence on HIS for diagnosis, suggestions, treatments and prescriptions for overall improvement in services and practices of healthcare institutions. These institutes have ample amount of data, but there is an acute shortfall in extracting useful information from this data.

One of the prerequisites for extracting useful information from medical records is to store them unabridged. Clinical notes contain rich and diverse sources of information but they cannot be stored capably in relational databases [2]. These databases are suitable for storing structured data, whereas Electronic Health Records (EHRs) are prominently semi-structured. Competent realization of HIS can not be accom-

plished by capitalizing only on traditional methods of data storage. Each database is delineated with different set of goals and is designed to solve distinct problems. Using only one database paradigm for all types of storage needs is evidently not a good strategy. 'Polyglot persistent solution' seeks to leverage highly specialized solutions for extracting useful information from the data.

We have developed an intelligent information integration solution — PolyglotHIS which uses multiple data-stores. PolyglotHIS frees the end-users from interfacing with each data-store individually by providing a uniform query interface. The key facet of proposed approach is the use of co-operative multi-agent architecture, along with the use of Datalog for declarative specification of contents and querying possibilities of underlying data-stores which also assists the query planning module in trimming trivial data-stores for a query. There is an urgent need to derive intelligence from the huge amount of stored data of the healthcare industry and PolyglotHIS is a step to address the same.

1.3 Research Gaps, Objectives and Methodology

Following major gaps are based upon the extensive literature survey and field study:

1. Traditional relational database based approach is not suitable for handling heterogeneous and scalable data.

2. No standard query language exists for NoSQL and NewSQL datastores, since each of these stores are designed differently.

3. There does not exist any middle-ware or adapters that provides layer over NoSQL data stores to allow a programmer interact with these data stores in SQL.

4. There is no single database system that can handle all sorts of data needed for an application. Different database systems perform better for specific kinds of data sets. For example, Cassandra is preferred for storing user data, MySQL is

good for storing transactional data and Memcached databases are well suited for session data.

5. Till date, no tool is available that automatically maps data present in a relational database to the NoSQL database. Transforming a relational data present in tables into different structures of NoSQL data stores is a cumbersome and tedious task.

6. There is no standardized NoSQL data store presently available that preserves needed integrity constraints.

7. There is a shortfall of the appropriate documentation available to guide the applicability of NoSQL data stores to different use-cases.

8. Schema extraction of NoSQL and NewSQL data-stores has not been explored extensively due to their schema-less data modeling approach.

Based on the above stated gaps, following objectives have been identified:

1. To study and analyze the existing databases such as relational, NoSQL, NewSQL etc.

2. To propose a novel framework for querying multiparadigm database which makee use of two or more classes of databases.

3. To verify and validate the proposed framework against conventional databases for efficiency.

To achieve the first objective, a comprehensive review was conducted to study various existing relational, NoSQL and NewSQL databases. Each class of NoSQL database is suitable for different use-case. For example, if requirement is to simply store and retrieve non-transparent data items using key, a key-value data-store like Redis will be preferred. If application needs to store records with lots of attributes, but generally asks for only a few certain fields and also, if the analytical functions are to be performed, then a column-oriented databases like HBase will be recommended.

The graph database, Neo4j, stores data in the form of nodes and relationships and provides various graph manipulation APIs that can be used directly by an application. Whereas, if the value associated with the key is needed to be searched and updated based on individual attributes within the value, a document database like MongoDB will be ideal. Storage of friend-of-a-friend type of data will be more natural in Neo4j graphs as compared to any relational database, which demands complex joins of multiple tables. There is no hard and fast rules for mapping system requirements to each class of NoSQL data-stores. To summarize:

- If an application simply stores and retrieves opaque data items that it identifies by using a key, then use a key/value store.
- If an application is more selective and needs to filter records based on non-key fields or update individual fields in records, then use a document database.
- If an application needs to store records with hundreds or thousands of fields, but only retrieves a subset of these fields in most of the queries that it performs, then use a column-family database.
- If an application needs to store and process information about complex relationships between entities, then use a graph database.

To achieve the second objective, a multi-paradigm database framework was designed in which multiple data-stores are involved, each dealing with different types of data set. The working of implemented framework can be summarized into the following number of steps: (a) an end-user (doctor/nurse) queries the integrated system, which further may require data from more than one underlying data-stores; (b) mediation layer composed of various co-operating agents processes the user request and determines how to split the asked query into sub-queries each pertaining to a different data-store; (c) sub-queries are executed on the underlying data-stores and individual results are returned to the reducer agent; and (d) results are merged and shown to the user in a desirable format by the graphical interaction agent.

Inter-operation between involved data-stores having different semantics, representation and querying mechanisms has been achieved with the help of various agents

which are software entities that chooses the best set of activities to be performed for reaching a particular goal as desired by the user, using artificial intelligence techniques.

The last objective, that is, the verification and validation of proposed framework was performed by analyzing and comparing the improved performance of proposed framework against existing approaches of data-integration. We have given a comparison of existing solutions dealing with multiple classes of NoSQL data-stores, along with supported data-stores, and their querying mechanism, and implementation approaches. Integrating data from disparate clinical data sources is an enduring research topic. Emergence of NoSQL databases have generated much excitement and interest The concept of a polyglot persistent application is still in its inception and, to the best of our knowledge, has not been implemented in the context of healthcare information systems. Although, few systems have been proposed recently that store data in (and query from) multiple data-stores discussed in Chapter 4, but none of them considers graph database, also existing solutions do not support join operation.

Experiments were performed to compare the performance of PolyglotHIS with systems that makes use of single database. The goal was to determine the overhead introduced by the multiple layers of the integration system. We have implemented four editions of the software; PolyglotHIS and other three editions of HIS using individual data-stores. Latency caused due to presence of multiple layers of PolyglotHIS is negligible and merges as the dataset increases. Evidently, in terms of functionality, HIS editions involving one data-store can accommodate greater variance as each data-store supports different range of features. In contrast, in case of PolyglotHIS, their functionality is limited by the common operations supported by all the underlying data-stores.

1.4 Research Contributions

This research contributes in the following ways:

- Modern data storage technology using NoSQL data-stores have been exten-

sively explored. An exemplary data-store of each of the four classes of NoSQL data-stores viz. Key-value, Column-oriented, Document and Graph-based, have been covered in detail under literature review.

- Each class of NoSQL data-store follows a specific data modeling and querying strategy. A thorough comparison has been presented, enabling an easy process of in selecting a particular data-store in accordance with the software application's storage requirements.

- Query languages of MongoDB (Document-based data-store) and Neo4j (Graph-based data-store) have been presented in details with exemplary queries and schema, developing a familiarity with the querying aspect of these data-stores.

- There has been a paradigm shift from relational to mixed data-stores to cater to the persistence needs of applications. In this research, a framework has been proposed that uses multi-paradigm databases i.e. various modules of an application can use specific data-stores to model their data that best match the semantic usages of data.

- Applicability and evaluation of proposed framework is demonstrated in context of Health-care Information Systems (HIS). HIS are multifarious in nature and, thus, are best candidates to be implemented by using the proposed multiple data-stores approach because one database definitely does not fit all the storage requirements of such complex applications.

- Multiple co-operative agents have been employed in the implementation of the mediation layer of the framework.

- The framework makes use of Datalog which is a declarative logic programming language, and is used in the proposed system in order to express schema of underlying data-stores in the form of a set of facts and rules.

- Performance of proposed framework that makes use of three different classes of databases have been compared with implementations of the application that

makes use of each one of the considered database systems individually. It has been observed that the latency due to presence of multiple layers in the framework is negligible and becomes insignificant as the dataset increases.

1.5 Organization of the Thesis

After giving an introduction to the thesis in Chapter 1, the rest of the thesis is structured as follows:

The *Chapter 2* presents a literature survey on database systems, explaining all three major classes of databases of importance and contemporary relevance. Firstly, relational databases are introduced, followed by discussing their limitations and inherent constraints posed by them. For handling today's un-structured big data, the importance of NoSQL data-stores have been already well established. NoSQL data-stores are quite different from the traditional relational database systems. Therefore their characteristics have been presented in detail. The latest class of database systems is the NewSQL data-stores, which includes hybrid of relational and NoSQL data-stores. NewSQL data-stores have been discussed briefly at the end of this chapter.

The *Chapter 3* elaborates on NoSQL data-stores, explaining all its four classes viz. Key-value, Column-oriented, Document and Graph-based. Pros and cons of each of the class along-with its potential use-cases is explained. One exemplary data-store of each class has also been discussed briefly. For each data-store, a code snippet in Python programming language has been discussed for performing basic CRUD operations with respective data-stores.

The *Chapter 4* discusses various data integration techniques primarily covering multi-base and federated databases, mediation based data integration, data warehouse and ontology-based integration. A detailed review of existing polyglot-persistent techniques has been presented in a tabular form based on a set of parameters, for quick comparison amongst them. At the end of the chapter, the need for polyglot-persistent applications in health-care data management has been under-

lined.

The *Chapter 5* describes the proposed multi-paradigm framework by explaining the individual components of the architecture. The proposed framework makes use of multiple co-operating agents especially at the mediation layer of the framework. Other two layers involved are the user layer and the data-store layer. The user layer contains only one agent, namely the graphical interaction agent, while the data-stores layer consists of three different data-stores and one partial result storage component. At the end, this chapter elucidates the need for such a polyglot-persistent framework in applications that deal with multiple modules.

The *Chapter 6* explores the creation and maintenance of Knowledge-base of Data (KBoD), which is the brain of the framework. Querying capabilities of the data-stores involved as well as the schemas of the data-stores have been expressed in form of Datalog facts and rules and stored in KBoD. The formulation of every query is performed with the help of KBoD. Before understanding the details of Datalog, a preliminary knowledge about first order logic and deductive databases have been covered in the initial sections of the chapter. Sample Datalog facts and rules that need to be present in KBoD for a sample data-set have been explained in detail.

The *Chapter 7* simplifies the understanding of the implemented framework and the concepts explained in Chapter 6 with the help of an example query. All three data-stores considered in the framework support different query languages, thus we have described basic syntax of their query languages using three trial queries. A specimen of data-set for both MongoDB and Neo4j is also presented as a pre-requisite to understand the working of the example query. Execution of the considered query is also shown diagrammatically for quick comprehension. Experimental analysis of the proposed framework is also illustrated using performance graphs.

The *Chapter 8* discusses various challenges and issues faced during the implementation of the framework. This chapter also gives concluding remarks on the research by highlighting the significant contributions of the work done. Future directions of the research work have also been presented towards the end of this chapter. The chapter concludes the thesis, by explaining briefly the outcome of each of the chapter.

Chapter 2

Databases: A Review

Until now, the term 'Databases' gets defaulted to 'Relational Databases', which are good at handling structured data. The data generated by Web 2.0 is primarily either unstructured or semi-structured. Thus, to handle this data a chain of non-relational data stores known as NoSQL databases have emerged and have got wide acceptance.

Unlike relational databases, NoSQL databases compromise ACID properties to provide high availability and scalability. Until recently the Industry have been built upon using ACID complaint softwares, hence they are now reluctant to non-ACID softwares. To bridge this gap, a new class of database, known as NewSQL databases that provides ACID properties of relational databases and scalability of NoSQL databases have emanated.

This chapter briefly review the three popular type of databases, starting from relational database to modern NoSQL and NewSQL databases. Timelines have been presented to visualize the emergence of tools under all three categories of databases. Characteristics, advantages and disadvantages of each type of database are also presented.

2.1 Relational Database

An organized collection of data is called database, while database management system is a software application responsible for storage and management of data. Each database is accompanied with an query language which is used to insert, update and delete data as well as to perform various database administration operations. In 1960s network model – CODASYL and hierarchical model – IMS were the most popular databases [3, 4]. In 1970 Edgar F. Codd's paper *"A relational model of data for large shared data banks"* proposed a relational model which is one of the most popular database model even today [5].

In mid 80s due to the surging popularity of object-oriented programming paradigm, object-oriented databases were introduced which also overcome the problem of object-relational impedance mismatch, but were later on merged with relational model itself – known as object-relational databases. After 1990s due to popularity of HTTP, the cost of posting and exchanging information became cheaper which led to the flooding of information. It was realized that traditional techniques of data storage will soon become stale and inefficient to handle such vast amount of unstructured and semi-structured data. A timeline visualizing emergence of various database tools since 1959 is presented in Table 2.1.

A database that stores data in form of relational tables is called relational database. Relational model was given by E. F. Codd in 1970 and is still in widespread use. He gave 12 rules that a database must obey to be termed as relational database, in which data is organized in the form of rows and columns within a table [6]. Each table may contain a *primary key* which is made up of one or more columns and is used for uniquely identify row of a table. Primary key of a table must be unique and not null, thereby maintaining ENTITY INTEGRITY. Tables are connected with each other using foreign keys, where a *foreign key* is a unique key in one table which is used to uniquely identify row in the joining table [7]. Foreign keys are also used to enforce REFERENTIAL INTEGRITY of data, for example a subject can be assigned to a teacher only if that subject exists in the Subjects table.

Relational databases provide quicker access to data with the help of *indexing*.

Table 2.1: Database Timeline

1959	CODASYL **(Network Model)**
1968	IBM's IMS **(Hierarchical Model)**
1970	**Codd's Relational Model**
1973	Berkley's Ingress
1974	IBM's System R
1978	Oracle
1980	Informix
1983	IBM's DB2
1984	Teradata
1985	**Object Oriented Databases**
1989	Microsoft's SQL Server
1994	Deductive Databases
1995	MySQL
1996	PostgeSQL
1998	InnoDB
1999	Netezza, XML Databases
2000	SQLite
2003	Greenplum
2004	**NoSQL Databases (Table 2.2)**
2006	**NewSQL Databases (Table 2.3)**

Indexes are usually created on combination of attributes. A query involving indexed attributes does not require full table scans and subsequently results in faster results. Performance of queries significantly increases when indexing techniques are applied on primary and foreign keys. *Join* is a read operation that allows to get data from more than one tables provided considered tables have some keys in common. Common keys usually involves primary key of one table and foreign key of another table. *Normalization* is the process of organizing columns and relations of the database such that data redundancy is reduced [8]. Relational tables are decomposed into smaller tables using the concept of primary and foreign keys. Aim of normalization is that operations such as addition, updation or deletion should be performed in one table only.

Data stored in relational database is managed using Structured Query Language (*SQL*)[9]. SQL is based on relational algebra and tuple calculus and supports both data definition language (e.g. create, alter, drop) as well as data manipulation language (e.g. select, insert, update, delete). SQL is undoubtedly the most popular and most widely database query language. SQL is primarily a declarative language, but is also extended towards procedural programming constructs known as PL/SQL. PL/SQL allows loop constructs and conditional statements along with variables and constants declaration and usage.

Relational databases provide transaction control using ACID properties. Each ACID property constitutes building block of relational database's transaction model. ***A****tomicity* ensures that each transaction is either done entirely or not at all, ***C****onsistency* enforces that execution of transaction takes database from one consistent state to another consistent state, ***I****solation* confirms that concurrent execution of transactions produces same results as if transactions are executed sequentially and ***D****urability* ensures that once a transaction is committed, data must persist even in the event of failures.

Relational databases have proved their mettle since last four decades. Major commercial relational database vendors are Oracle, IBM, Microsoft, SAP and Teradata. Open-source implementations of relational databases are also in widespread use e.g. MySQL, PostgeSQL and SQLite. ACID compliance of relational databases

makes it most suited for OLTP (On-Line Transaction processing) applications. Relational database have matured incredibly with their age and provides all the advanced features expected from a database management system including logging, reporting, query optimization, recovery mechanism etc. SQL have also advanced with age in terms of features. It is very easy to find help in any context in case of relational databases. Finally, relational databases are supported by everyone and everything.

2.1.1 Shortcomings of Relational Databases

Relational databases were designed for structured data. But a large percentage of digital information floating around the world is in PDF, HTML and other types of formats which cannot be easily modeled, processed and analyzed using RDBMS. Natural text can not be easily captured in form of entities and relationships. The major problem is that the legacy tools require data schema to be defined a priory to the data creation. In today's world, where Internet has become an important part of the common person's life, deciding on rigid pre-defined schema is unrealistic. Schema evolves as users start using the application and as the information to be dealt with, varies according to user's requirements. Structure of data also changes as information is gathered, stored and processed. Also, all the available rich information cannot be forcefully made to fit in the tabular format of relational databases. This problem was also faced by object-oriented databases under the name "Object-Relational Impedance Mismatch" [10]. This mismatch occurs when an object is molded to fit into relational structure.

Above stated problems of storing semi-structured information, object-relational impedance mismatch and schema evolution originated because of the changes in usage pattern of databases since the time the idea of relational databases were conceptualized in 1970s. Relational databases were not designed keeping in mind sparse information and scalability, where scalability is the ability of a system to handle growing amounts of data. Their data model was based on single machine architecture and was not designed to be distributed. Today all softwares are developed expecting a large user-base, which was not the case in 1970s.

Few shortcomings of relational databases due to which NoSQL databases emerged are enlisted below :

1. *Structured Vs Unstructured Data* – Relational databases were designed to store structured data. But majority of data available today is either semi-structured or un-structured. Attempt to store semi-structured data in relational databases produce sparse tables, where rows contains large number of NULL values corresponding to optional attributes.

2. *Schema Evolution* – Relational databases demand that schema of an application must be determined and fixed a-priori, this inflexibility in data-modeling limits schema-evolution. Schema evolution refers to the problem of incorporating changes to the database schema of an application as per the requested change.

3. *Scalability* – Relational databases were not designed to be scalable and go beyond one server, because they were developed in an era when user-base was very limited and Internet was also in its inception.

4. *Flexibility* – Relational databases follow "one size fits all" approach, where any type of data is forced to fit in row-column format of relation (table).

5. *Normalization* – Performance of system degrades when the database system consists of lots of normalized tables. More is the number of tables, more time will be spent in indexing, joining and other similar operations.

6. *Joins* – Storing large amount of data in normalized fashion will result in significant reduction in performance due to expensive join operations required to access data.

7. *Connected Data* – Data generated from semantic web is highly connected and if stored in relational database will result in a series of expensive join operations.

8. *ACID properties* – Adherence to strict ACID properties in case of relational databases, limits availability as well as scalability of the system.

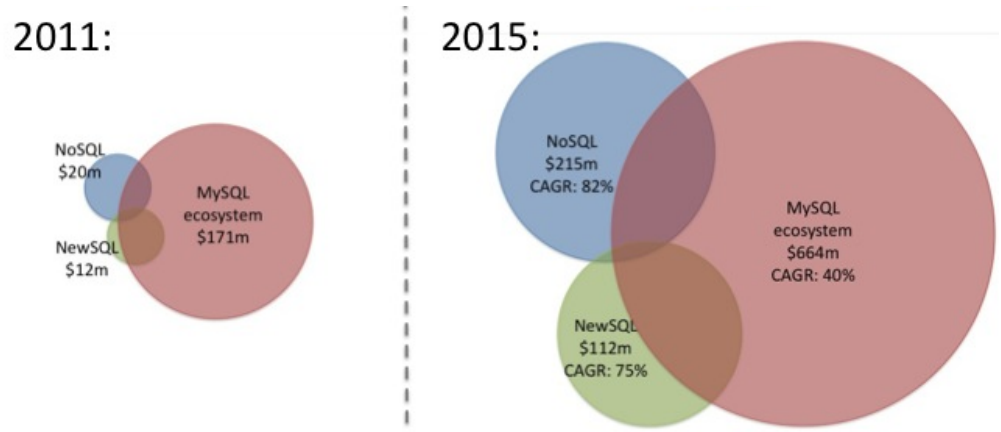

Figure 2.1: MySQL, NoSQL, NewSQL Revenue [11]

Above mentioned problems of relational databases have lead to the development of a new class of databases, popularly known as NoSQL databases. NoSQL databases were designed considering the requirements of modern applications, thereby are attempted to overcome all of the shortcomings of relational databases. Unlike relational databases which are suited for structured data, NoSQL databases handle all the three types of data (structured, un-structured and semi-structured) very well. NoSQL databases are schema-less databases and provide high scalability horizontally. NoSQL databases come in four flavors providing flexibility to the developers to choose from a variety of databases. NoSQL databases also do not support normalization, joins and ACID properties.

NoSQL solutions should not be thought of as a replacement for RDBMS, instead as a complementary product for handling issues of scalability and complexity. Non-relational databases provide many enhancements over traditional relational databases such as increased scaling across commodity servers or cloud instances, non-adherence to rigid schema for inserting data and hence ease in capturing of different type of data without much changes at schema level. These NoSQL databases may require additional storage, since data is de-normalized, but results in overall improvements in performance, flexibility and scalability.

The 451 research group published a report which states that although impact of

NoSQL and NewSQL databases on MySQL is not huge as of now, but they certainly pose long term threat due to their increased adoption in new projects. The key findings of report is shown in Figure 2.1. Next two sections briefly discusses NoSQL and NewSQL data storage solutions and explains how these databases are different from relational databases.

2.2 NoSQL Databases

Although relational databases have matured very well because of their prolonged existence and are still good for various use cases, unfortunately for most of the today's software design, relational databases show their age and do not give good performance especially for large data sets and dynamic schemas. We live in a world, where domain model is constantly changing both during development phase and even after deployment. These changes in requirements along with various other reasons described above, led to the development of non-relational databases known as NoSQL databases. Popularity of non-relational databases can be very well imagined by the fact that many universities have started teaching about these data stores as part of their curriculum [12].

NoSQL which stands for Not Only SQL, is the term most commonly used to cover all non-relational databases and it . There is much disagreement on this name as it does not depict the real meaning of non-relational, non-ACID, schema-less databases, since SQL is not the obstacle. The term "NoSQL" was introduced by Carlo Strozzi in 1998 as a name for his open source relational database that did not offer a SQL interface [1]. The term was re-introduced in October 2009 by Eric Evans for an event named *no:sql(east)*, organised for the discussion of open source distributed databases [2]. The name was an attempt to describe the increased number of distributed non-relational databases that emerged during the second half of the 2000s.

[1] NoSQL (http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page)
[2] NoSQL (https://nosqleast.com/2009/)

Table 2.2: NoSQL Timeline

2004	Allegrograph
2005	CouchDB
2006	Google's BigTable
2007	Amazon's Dynamo, Neo4j
2008	Cassandra, MemcacheDB
2009	MongoDB, HBase, Voldemort, Riak, Redis
2010	OrientDB, Infinite Graph
2011	HyperDex, Oracle's NoSQL
2014	Accumulo

Increasing number of players dealing with WWW started recognizing the in-efficiency of relational databases to handle huge amount of diverse data generated by the introduction of Web 2.0 applications. Google was the first organization to lead this movement by introducing BigTable [13] in 2006 followed by Amazon's Dynamo [14] in 2007. Influenced by adoption of non-relational databases by these big firms most of the organizations started developing their own NoSQL data stores customized to their requirements. Most of today's popular NoSQL data stores have adopted ideas either from Google's BigTable or Amazon's Dynamo. Those inspired by BigTable are categorized as column-oriented or wide-table data stores and others which are descendants of Dynamo are termed as key-value based data stores. There are two other categories also, namely document and graph-based databases. These four classes of NoSQL databases deal with different types of data and hence are suitable for different use cases. Each has its own advantages and disadvantages in a particular context. Various popular NoSQL databases are shown in Table 2.2 according to their release year. Figure 2.2 shows the landscape of different classes of databases given

by the 451 research group.

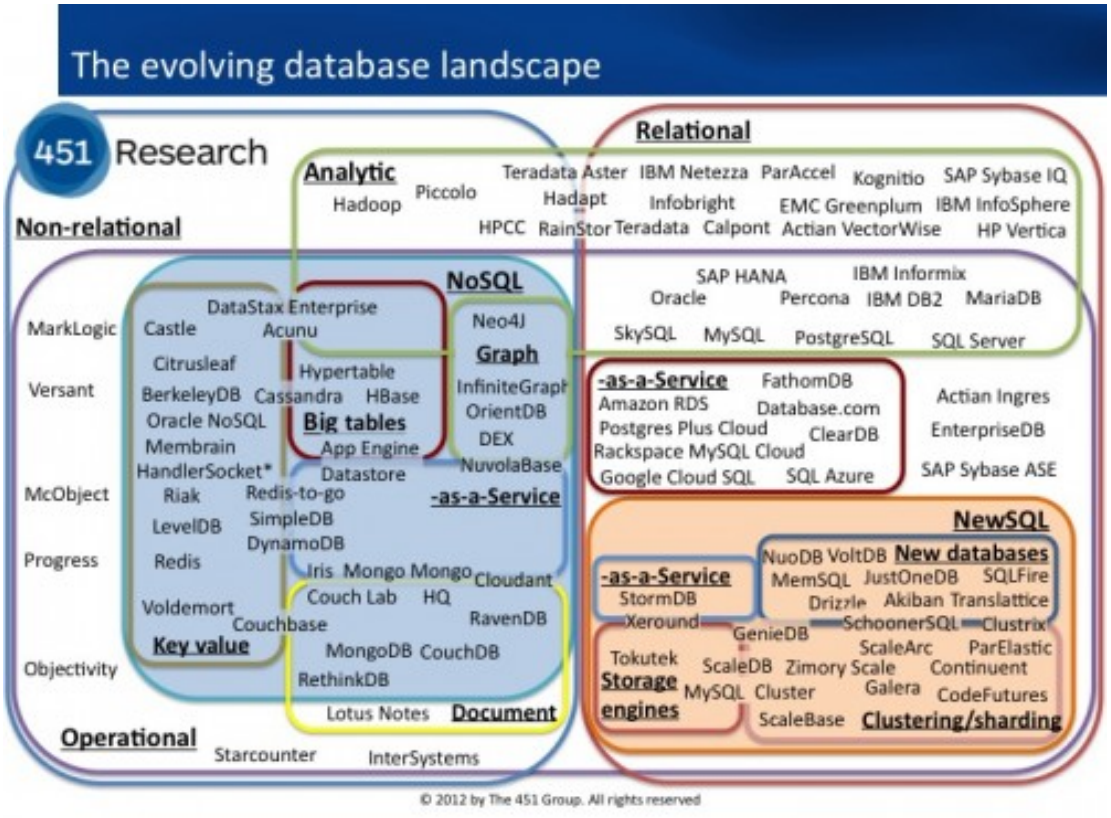

Figure 2.2: Database Landscape [11]

2.2.1 NoSQL Characteristics

There is no fixed definition of NoSQL databases, but most of them conform to the following properties: *Schema-less, Scales horizontally, Join-less, ACID-less (BASE), Non-Relational, Does not use SQL, Open-source (mostly) and Supports big data.*

Schema-less

All modern data-stores do not require schema of the application to be fixed prior to the design and development phase. Instead, they provide flexibility in schema upda-

tions technically known as *Dynamic Schema Evolution*. Unlike relational database, each class of NoSQL database provides flexibility in terms of schema. For example, in document database different documents may have different fields. Similarly in Key-value database, values may have entirely different number and type of fields. Same is true for the graph database as different nodes and edges may have different properties.

Scales horizontally

Scalability is one of the most discussed issue today, since web applications have got enormous popularity. Scalability can either be achieved vertically or horizontally. Vertical scalability, also known as scaling up is easier to achieve as compared to horizontal scalability also known as scaling out. As the name suggests, scaling-up means addition of resources to a single node and scaling-out means adding more nodes to a system. Horizontal scaling provides more flexibility as commodity servers or cloud instances can be utilized [15]. Traditional databases rely on vertical scaling where as recently evolved non-relational databases use horizontal scaling [16].

Sharding is the process of fragmenting data across clusters based on some predefined criteria. Shards are typically partitioned based on the value of row key, consequently query on a particular value need not traverse all nodes. Sharding can improve the performance of data retrieval significantly by accessing the preferred node without putting additional load on other nodes. Along with horizontal partitioning, column groups are formed to achieve vertical partitioning. Sharding when accompanied with replication, improves fault-tolerance of the node, achieving graceful degradation. Most of the popular data-stores today such as MongoDB provides built-in support for automatic sharding and replication.

Does not support join

Joins being expensive operation should be avoided especially in the case of large data. Joins are a by-product of normalization, since the process of normalization re-

sults in multiple small tables which need to be joined for querying purposes. NoSQL databases do not use normalization hence need not to support join operations. If required, code to join data in NoSQL databases is written at application level. Undoubtedly, de-normalization cause duplication of data but it provides significant performance enhancements [17].

Provision of joins in relational databases require strong consistency and fixed schemas. Avoiding joins in NoSQL databases have made flexibility in schema and horizontal scalability easy.

Non-ACID – BASE

Unlike relational databases which strictly follow ACID (Atomicity Consistency Isolation Durability) properties, NoSQL datastores follow relaxed approach towards these properties. Eric Brewer provided CAP theorem in 2000 which states that it is not possible to provide all three properties namely *Consistency, Availability and Partition Tolerance* at the same time in an application [18]. CAP theorem was later formally proved by Gilbert and Lynch [19]. *Consistency* is one of the ACID properties and ensures that each operation is either done fully or not at all. *Availability* of a system means that the system is up and running at all the times. Availability of a system is dependent on its reliability. *Partition Tolerance* refers to the ability of a system to continue working in the presence of communication errors among subsystems due to network partitioning. As shown in Figure2.3, databases are classified on the basis of which two properties of CAP theorem they provide.

BASE (Basically Available, Soft-state, Eventually consistent) follows that application should always be available, not all sites contain the current copy of data but eventually everything will be consistent [20]. Soft state refers to the state when multiple sites contain different copy of data due to asynchronous replication which is essential for availability. At some point of time, whole system will become consistent which is termed as eventually consistent.

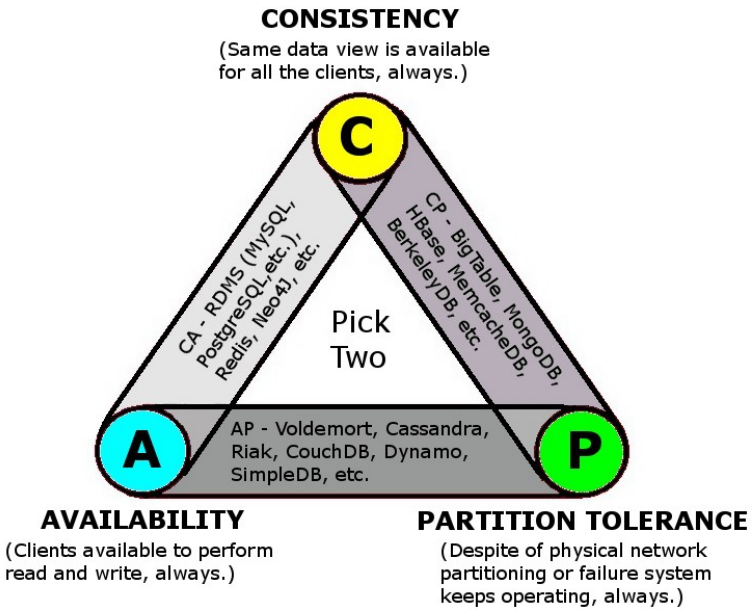

Figure 2.3: CAP Theorem

Non-Relational

NoSQL databases are also popularly known as non-relational databases attributed to the fact that they do not store their data in relational/tabular format. Data generated by Web 2.0 websites is largely semi-structured or un-structured, which if forced to be stored in relational model will result in sparse rows i.e. rows containing lots of NULLs. NoSQL databases thus provide programmer with a flexible model to store large amount of semi-structured and un-structured data.

Does not use SQL

NoSQL databases are classified into four classes and each class further encompasses number of datastores, each developed for specific requirements. Each NoSQL datastore comes with its own specific query language. This characteristic of NoSQL datastore is more towards disadvantage side. SQL provided uniformity in querying all relational databases, but in case of NoSQL datastore due to lack of one uniform

query language, developers have to learn syntax of each datastore to work with it. Help and manuals, in the case of SQL, are available in plenty due to its maturity and widespread use, which is not the case for young NoSQL datastores.

Open-source

One of the most bright side of the NoSQL movement is the availability of a number of open-source databases. The tool box of the developer has increased significantly both in number and variety.

Supports big data

Necessity is the mother of all inventions. NoSQL databases were invented because they were required for handling big data. Traditional databases designed in 1970s can not cope up with the requirements of data being generated today in all the three aspects – *Volume*, *Variety* and *Velocity*. NoSQL databases are inherently scalable (to handle Volume), and can easily store data in different formats (to handle Variety) due to their schema-less nature. Major NoSQL databases provide built-in support for Map-Reduce framework to process and analyze big data.

2.3 NewSQL Databases

NewSQL databases bridge gap between relational and NoSQL databases by providing ACID properties and SQL of relational databases, and schema-lessness and scalability of NoSQL databases. NewSQL database was first mentioned in 2011 in a research report of 451 research group [11]. NewSQL databases offer a middle way to the industry, since these databases are designed for data management in cloud environments while keeping SQL for interfacing and retaining the popular ACID properties. Leading NewSQL databases are Google's Spanner [21], VoltDB [22] and NuoDB [23].

Table 2.3: NewSQL Timeline

2006	Clustrix
2007	H-Store
2008	NuoDB
2010	VoltDB, SAP Hana
2012	Google's Spanner, MemSQL, CouchBase, Aerospike
2013	FoundationDB

Table 2.3 shows various popular NewSQL databases in the order of their release year. NewSQL databases have re-architectured the database management system from scratch to meet the changing requirements of software and data, while keeping in mind the application already designed using relational databases. Hence, NewSQL databases are best suited for legacy applications desiring to adapt themselves to new trends in data growth.

Extent of support for SQL as query language varies among various NewSQL datastores, for example NuoDB and Clustrix offer the maximum compliance with SQL. VoltDB restricts the use of *having* clause and mostly interacts using stored procedures. This class of database is specifically suited for applications performing transactions that involve manipulation of more than one objects. A detailed comparison of all the four classes of NoSQL data-stores and NewSQL data-stores is presented by Grolinger et al. [24].

Relational database systems keep the logical model separate from physical representation and processing strategies. They are good at handling facts and extracting reliable information from collections of facts. While implementing relational databases a lot of attention is paid on developing conceptual, logical and physical data models. New technologies such as XML, NoSQL and NewSQL databases have put doubts on the usefulness of data models. For example, storing data in databases

as XML blobs and documents in case of document-oriented NoSQL databases try to eliminate the need for data modeling. But to describe any data structure, rules can best be described as a data model. Although, these new generation of databases are schema-less but the importance and requirement of data model will always remain to understand and demonstrate the storage of data. Unlike relational databases where modeling is decided by the structure of data, while modeling NoSQL databases the types of queries those will be executed on data are also kept in mind. In other words, design theme of relational database are focused on *answers* whereas NoSQL and NewSQL databases are focused on *questions*.

Availability of different classes of databases, bring the challenge of careful selection of one class for your application. Pros and cons of each datastore need to be analysed and then aligned with the requirements of the software to be developed. Furthermore, Sadalage and Fowler [1] warrant the use of more than one type of database within an application for different purposes. NoSQL data-stores expose four data-stores namely key-value, document-oriented, columnar and graph oriented data-store [25]. Chapter 3 discusses in detail all four classes of NoSQL data-stores, giving one popular example of each class.

Chapter 3

NoSQL Data-Stores

Leading reason for choosing emerging NoSQL data stores over mature and rigid relational databases is that they fit to data processing strategy more closely. NoSQL solutions are not necessarily a replacement for RDBMS, but a complement to handle issues of scalability, complexity and flexibility. Also, since most NoSQL databases are schema-less, development gets easier and faster.

Classification of NoSQL databases is done based on which database can best store associative schema-less or document or column-oriented or graphical data, each class dealing with specific use-cases. The broader aim is to enable programmers to choose database that models the data in the best possible way.

All four classes of NoSQL databases are presented in this chapter, along with brief description of one popular database of each of the four classes. For easy understanding, a case-study of heath-care information system is represented diagrammatically to illustrate the differences in modeling of data in each class.

3.1 Key-Value Data-store (KVS)

Key-value databases have a very simple data model — data is organized as an associative array of entries consisting of key-value pairs [26]. These databases can be visualized as relational databases having multiple rows and only two columns — key and value. Each key is unique and is used to retrieve the values associated with it. Data type of key is limited to string, but there is no limit on type or length of value to be stored. These databases store all data relevant to an item together, which increase duplication of data but improves scalability due to absence of joins.

Unlike relational databases, referential integrity is taken care at application level instead of being enforced at database level. Key-based lookups result in lesser query execution time. Moreover values can be anything like objects, hashes etc., resulting in flexible and schema-less model appropriate for today's unstructured data. These databases are highly suitable for applications where schema is prone to evolution. In order to understand KVS, consider an operation to store details of a patient and his/her treatment profile in a KVS based data-store. Figure 3.1 shows how KVS performs the data storage.

Key-value data-stores are further of two types depending on where data is being stored — *in-memory* and *persistent*. Example of in-memory key-value stores are Memcached [1] and Redis [27]. Other type of data-stores that stores data on disk are called persistent key-value data-stores, examples are Voldemort [28] and Riak[2]. Voldemort and Riak also provide support for Map-Reduce querying mechanism. These data-stores lack security as they lack Authentication, Authorization and Auditing [29].

Use-cases: Content providing applications, Object Caching, Session management, Storing user profiles and shopping cart etc.

Pros: Simple, Scalable, Suitable for cloud environment, Programmer-friendly data structures and schema-free.

Cons: Integrity constraints at application level, Limited analytics capabilities,

[1]Memcached (http://www.memcached.org/)
[2]Basho Technologies, Inc. Riak (http://wiki.basho.com/Riak.html)

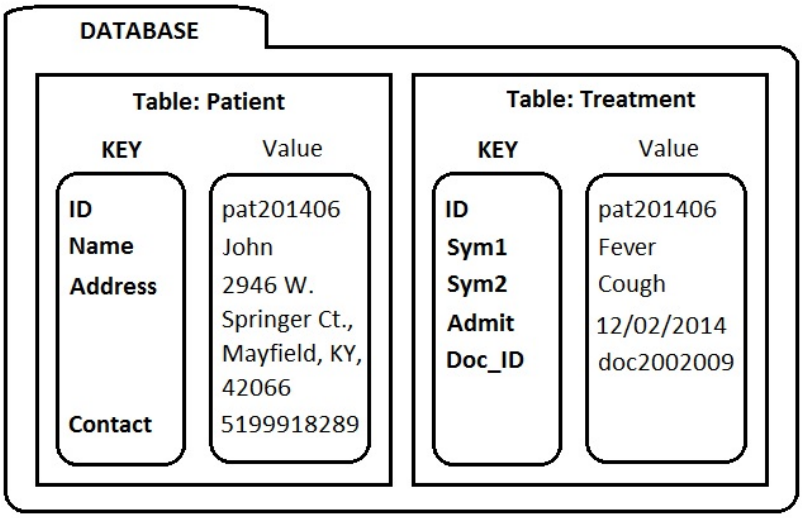

Figure 3.1: Data storage in Key-value data-store

Not suitable for storing connected data and incapable of querying by value.

3.1.1 Redis

Redis stands for REmote Dictionary Server and is an open-source data-store. It is an in-memory key-value data-store, writing back to disk only for persistence [30]. It is also called data structure server as its keys can support strings, lists, hashes, sets and sorted sets[3]. All these five data structures are discussed in detail in the next section.

3.1.1.1 Data Types in Redis

Redis exposes five data structures or data types and each one of them has its own specific set of commands for interaction. Some operations and scenarios are discussed in this section in order to illustrate the usefulness of these data types. Every data type has its own specialty and must be chosen carefully to exploit it to the fullest.

[3]Data Structure Server (http://en.wikipedia.org/wiki/Redis)

Strings are the most basic and primitive data type available in Redis. Strings in Redis are binary-safe, because they store everything in the form of raw data without any specific format and thus can store anything like integers, strings, JSON objects, etc. Having the basic key-value structure they provide a large number of operations in order to exploit them to the fullest. Strings are very easy to use when we have to maintain sessions and needs to set time to expire for the given key or in situations where unique IDs need to be generated.

Strings fall short when we have to store something in form of columns and value type structure similar to relational model's attribute-value storage. Every data type has its own peculiarity as said earlier and for this particular situation, Redis has just the right thing: *hashes*.

Hashes are very easy to operate as they provide various operations to exploit both value as well as fields. Hashes give outstanding performance up-to few hundred fields and thus can efficiently store numerous such objects without losing the performance factor [31]. Strings as discussed store everything as a serial object which renders them useless whereas hashes with their filed names do wonders. One of the many features of hashes helps to keep a check on the redundancy of data and thus prevents setting value to a field for which one has already been assigned and that too in constant time complexity.

Hashes can be implemented wherever we have relational model kind of structure to store. Hashes' filed can be closely related to column-name in relational model. Strings can store data as serial objects and hashes do the same job, although a field was added to enhance the features and usability. If we want to store data as an array of objects corresponding to some key, then lists are the data types that are most suitable.

Lists in Redis are implemented as linked lists which means that objects can be added in constant time to either end of the list. Lists have some unique features of its own and thus can cater to the needs wherever queues are needed to be maintained. In addition to their implementation as queues, lists are of great use in other situations like maintaining the list of recent 'n' visitors, keeping chat log for a messenger, etc. But when we have to perform operations pertaining to presence of

some record or getting all the instances of the record in the database then the data type with associative array storage like lists does not help much. One would think of mathematical sets and to search all such instances of the record, intersection of all the sets comes into mind. This is what particularly sets excel in and differentiates them from all other data types.

Sets are a collection of strings stored in an unordered fashion. Most promising feature they provide is that they store unique values only, thus reduce the overhead of checking the whole data while inserting a new value [32]. As the name suggests sets relate to algebraic sets only and various set operations like union, intersection etc. can be performed on them to fetch desired results. So, when our application needs comparison related operations, sets prove to be most optimal and efficient resort.

Some more operations are also available that can help to exploit sets to the maximum. Sets may store and pool the data together as intended perfectly but it falls short on some aspects too. It stores data as unsorted strings, giving preference to none. Let us assume a situation where strings in the set are to be sorted in some given order or priority and need to be considered accordingly. What to do in such situations? These situations need a data type which is more powerful and advanced than sets and Redis has another data-type called Sorted Sets to handle these situations.

Sorted Sets are the most powerful weapon in Redis's arsenal. Sorted sets are similar to sets but with an additional field score to keep rank of the values. Like sets they also hold unique values only. By default they store the value in increasing order of the score. Every time a new value is added, it is inserted to the location according to its score. This makes fetching of data sorted by the score or the rank easier. Many applications can be modeled using this data structure.

To understand and implement Redis using Python, *redis-py* repository is used which helps building applications using both Python and Redis together. To use Redis via Python a connection is needed and to do that Redis needs to be loaded in Python environment. A code snippet demonstrating how hashes and sorted sets of Redis are operated using python is shown in Figure 3.2.

```
import redis
redisconn=redis.StrictRedis(host='localhost',port=6379, db=0)
# HASHES
# adding patients to respective departments
redisconn.sadd("ent:patients","pat201406", "pat201407")
redisconn.sadd("dental:patients","pat201408", "pat201401")
check=redisconn.sismember("ent:patients","pat2014070008")
if not check:
    print "not a member"
# transferring a patient
redisconn.smove("ent:patients","ortho:patients","pat201407")
# finding patients common in departments
print redisconn.sinter("ent:patients","dental:patients", "ortho:
    patients")

# SORTED SETS
# adding stage of patient's illness
redisconn.zadd("hrms:onco:patstage", 3,"pat201407", 4,"pat201406", 1,"
    pat201408",4,"pat201401")
# patients sorted in descending order of stage
print redisconn.zrevrange("hrms:onco:patstage",0,-1,"with_score")
# no. of patients with illness between given range of stages
print redisconn.zcount("hrms:onco:patstage",2,4)
# strict redis implementation takes increment as the last arguement
redisconn.zincrby("hrms:onco:patstage", "pat201406",1)
```

Figure 3.2: Redis hashes and sorted sets using python

Redis and its data types can handle almost every type of data simply and efficiently provided the right data type is chosen for the same. Each data type can be exploited to do some particular type of job in best possible way. Although one could use other data types as well for the same job but they might not be so efficient for the given job. So, it all relies upon the right choice of data type for your storage needs.

3.2 Column-Oriented Data-store

Also called *Wide Table data stores*, are primarily designed to address following three requirements — huge number of columns, sparse nature of data and frequent changes in schema. Column-oriented database systems perform better than traditional row-oriented database systems, especially for analytical purposes as used in data warehouses, decision support, and business intelligence applications [33, 34]. A row-oriented database must read the entire row in order to access the required column where as columnar storage will read only those columns that are asked for [35]. As a result, analytic and business intelligence queries generally read significantly more data than is needed to satisfy the request, which causes unnecessary input-output as shown in Figure 3.3.

These extensible record stores have been motivated by the success of BigTable,

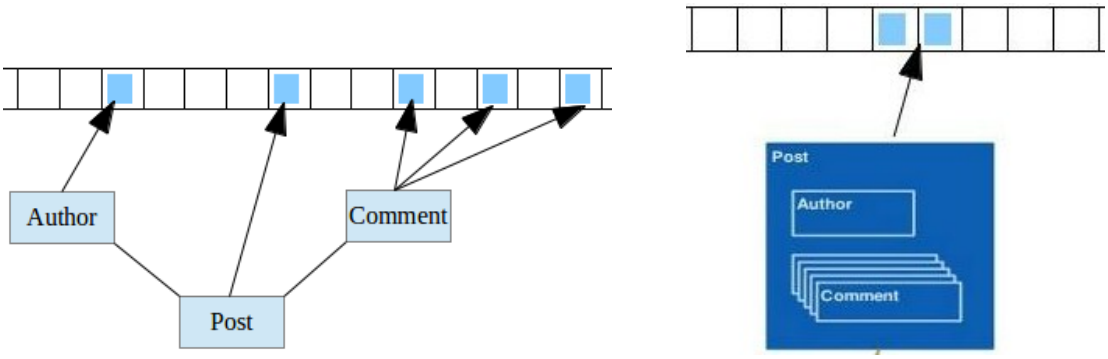

Figure 3.3: Row Vs Column-oriented storage

which is a NoSQL data store introduced by Google in 2006 [13]. Popular open source column-oriented databases are Hypertable [36], HBase [37] and Cassandra [38]. Hypertable and HBase are derivatives of BigTable where as Cassandra takes its features from both BigTable and Dynamo.

Most of the columnar databases are also compatible with MapReduce framework, which speeds up processing of large amount of data by distributing the problem on large number of systems [39]. Also, in row-oriented storage design each column deals with multiple data types and infinite range of values, thus making overall compression less efficient as compared to columnar databases [40].

A column family is a container for rows, analogous to the table in a relational system. Each row in a column family is referenced by its key. A column family can contain columns or super columns, where a super column is a dictionary, it is a column that contains other columns. Columns and super columns in a column database are sparse, meaning that they take exactly 0 bytes if they do not have a value in them. All the data in a single column family resides in the same file.

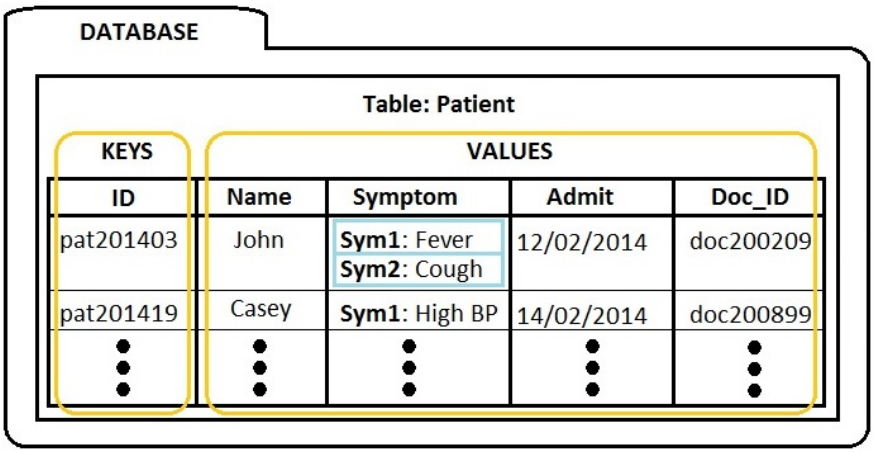

Figure 3.4: Data storage in Columnar data-store

Columnar databases consists of three elemental things, a unique name that refers to the column, a value field to store the data that can be of various types like ASCIItype, LongType, etc. and a time-stamp to validate the content. At the column level they

are also key- value pair. This approach can be well implemented in clinical data storage where the data varies from a patient to patient and so does the number of columns. For instance the symptoms shown by a patient can never be fixed and varies from patient to patient and hence there is need for columnar approach. Fig. 3.4 represents columnar method to store such data.

Column-oriented data stores are the preferred method for storing time series data in many applications in capital markets [41]. For example, they are excellent choices for storing tick data from stock exchanges, where tick data refers to market data which shows the price and volume of every print, also often includes information about every change to the best bid and ask. Performance gain is attributed to the fact that only those attributes that are required by the query are read from disk/memory. A detailed comparison between row and column stores in terms of performance and query execution is available in [35].

Use-cases: Analytical querying, Blogging websites, Event logging, Content Management Systems, Maintaining counters.

Pros: Flexible, Scalable, Efficiently handles random read operation, Supports Map-Reduce, Provides high write throughput,

Cons: Integrity constraints at application level, Not good at handling relationships, Ad-hoc querying needs revamping of column-family design.

3.2.1 HBase

HBase is an open source, non-relational, distributed database modeled after Google's BigTable and is written in Java [37]. It is developed as part of Apache Software Foundation's Apache Hadoop project and runs on top of HDFS (Hadoop Distributed Filesystem) [42, 43], providing BigTable-like capabilities for Hadoop. HBase is basically a map as analogous to associative array but unlike other map implementations, key-value pairs are stored strictly in an lexicographical order with the lowest order appearing first in the table. This feature of storing values according to sorted keys ensures that values of like keys are physically stored approximate to each other which results in better performance while accessing data from large and distributed

systems. This feature also enables the possibility of performing range queries on keys.

Each row of data have a unique identifier called row key and a number of columns. The number of columns can differ among the rows resulting in capability of storing sparse data. This characteristic also allows dynamic and easy addition of columns to specific rows. Column families comprise of any number of columns sorted by their name, they groups data (columns) of same type. A column family may consist of one or more columns. Different column families may contain different number of columns or may contain no column at all. It is easy to add new columns within a column family, whereas adding a new column family or modifying an existing one is very difficult and expensive task. Number of column families are decided at the creation of database. Each row can have multiple columns and column families. Rows do not have null columns or pre-defined column width.

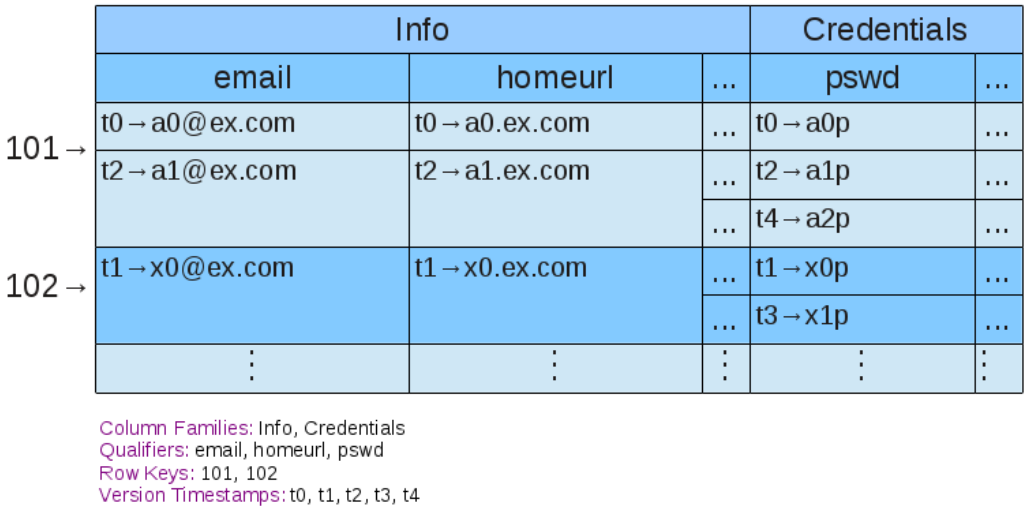

Figure 3.5: Column-families and Versioning in HBase

One of the most important advantages provided by HBase is versioning. In most of the databases only one value is stored for each attribute, therefore every update over-writes the previous value [44]. HBase allows storage of multiple values for any particular attribute using versioning, as shown in Figure 3.5. Each version of the

value is differentiated from others via time-stamp which is stored along with the data. Timestamps can be explicitly defined by the user while inserting data. Data is stored in reverse chronological order, which means value with highest time-stamp is stored on the top. This is the most recent value and will be returned if asked only for value. If time-stamp is also mentioned, while asking for the value then value corresponding to that time-stamp or with lesser value of time-stamp will be returned.

Data stored in HBase can be queried using HBase shell. HBase has its own web-based interface as part of its installation. There are many other open source projects aimed at providing visual editors to query HBase data, for example Toad for Cloud[4] which is a free SQL-based tool with the familiar look and feel of Toad that enables users to use SQL with a host of emerging non-relational databases including HBase, Crux[5] which provides a simple web based graphical interface to access HBase, query data and create reports, HBaseXplorer[6] which is an Java desktop application for managing and exploring HBase database and HBaseExplorer[7] that allows to manipulate and explore HBase instances. Querying options in HBase are very limited as of now especially in HBase shell, as compared to other established databases. HBase can also be interacted using python library *HappyBase* which internally uses Python Thrift library. An example code snippet is shown in Figure 3.6.

Unlike relational databases, where schema is fixed and common for all rows, NoSQL databases like HBase are schema-less resulting in removal of constraints over list of keys. Value corresponding to a number of keys can be a complex structure in itself unlike atomic values in case of traditional relational databases. Addition of a new key-value pair can be done at any time without affecting the remaining database. Row key in column-oriented databases acts as primary key of relational databases.

[4]http://toadforcloud.com/index.jspa
[5]https://github.com/sonalgoyal/crux
[6]https://github.com/bit-ware/HBaseXplorer
[7]http://sourceforge.net/projects/hbaseexplorer/

```
import happybase
connection = happybase.Connection('localhost')
connection.open()


connection.create_table(
    'Patient',
    {'Info': dict(max_versions=10),
     'Credentials': dict(max_versions=1, block_cache_enabled=False),
    }
)
table = connection.table('Patient')
table.put('101', {'Info:email': 'a1@ex.com',
                  'Info:homeurl': 'a1.ex.com',
                  'Credentials:pswd': 'a2p'})
row = table.row('101')
print row['info:email']  # prints 'value1'
for key, value in table.rows(['101', '102']):
    print key, value  # prints row key and data for each row
for key, value in table.scan(row_prefix='10'):
    print key, value  # prints 'value1' and 'value2'
row = table.delete('102')
```

Figure 3.6: Interacting with HBase in Python

3.3 Document Data-store

A document database is used to store, retrieve and manage semi-structured data. In this database, data is stored in the form of documents. Documents are grouped together in form of collections. These databases are flexible in nature as different

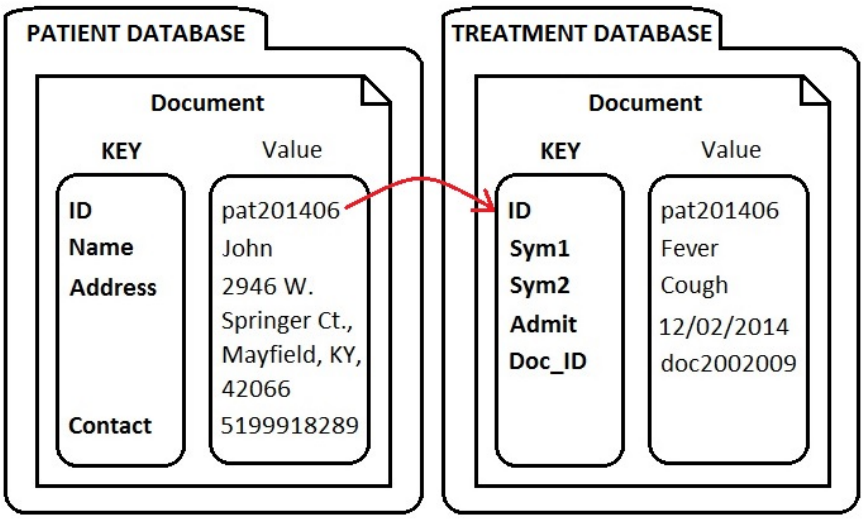

Figure 3.7: Data storage in Document-oriented data-store

documents can have different fields, also any number of fields can be added to the documents without wasting space by adding same empty fields to the other documents in a collection. Document can be in any format like XML (eXtensible Markup Language) [45], YAML (Yet Another Markup Language) [46] and JSON (JavaScript Object Notation) [8]. Document-oriented databases are one of the categories of NoSQL databases that are appropriate for web-applications which involves storage of semi-structured data and execution of dynamic queries.

Document data-stores are inherently key-value data-stores, but here values are stored in a structured format providing transparency and enable indexing and look-up on values. Operation on these data-stores are no more limited on keys only. Document data-stores enable us to store arbitrarily complex data such as trees, dictionaries, collections very naturally as shown in Figure 3.7. One document can refer to other documents by referring their object_ids, but there is not in-built referential integrity like relational databases. Like previous data-stores discussed in Section 3.1 and 3.2, document data-stores are not good at handling data containing relationships.

[8] http://json.org/

Practical usability of these databases can be guessed by the fact that there are more than 15 document-oriented databases available, of which widely used are MongoDB [47], CouchDB [48] and RavenDB [49]. CouchDB and RavenDB store data in JSON format, where as the most popular document database MongoDB stores data in BSON notation where BSON is Binary JSON that enables binary serialization on data. Advantage of storing data in JSON or BSON representation is that it is easy to map object structures of most of the programming languages directly into this representation, no mapper/translators are required [50].

Use-cases: Blogging systems, Content Management Systems, Real-time analytics, Mobile applications, Inventory management, Shopping cart and Logging events etc.

Pros: Powerful indexing, Good at handling complex data structures, In-built support for Map-Reduce, Schema-less, Scalable, Fast writes, High availability, Sharding Lesser workload on DBA and Suitable for agile development.

Cons: Not good at handling data containing relationships, Data duplication, No transactional capabilities, Searching and Not good at performing complex transactions.

3.3.1 MongoDB

MongoDB is the most popular document database. It is designed to be able to face new challenges such as horizontal scalability, high-availability and flexibility to handle semi-structured data. MongoDB stores data in the form of documents which are grouped together in collections [51]. Compared to relational databases, collections correspond to tables and documents to records. Different documents can have different fields, also any number of fields can be added to a document without wasting space by adding same empty fields to the other documents in that collection. Unlike relational databases where every record in a table has the same number of fields, documents in a collection can have completely different fields.

```
> db.Patients.find().pretty()
{
"_id":"OCLS1326",
"Name":"P1",
"Gender":"M",
"DOB":"18/07/1976",
"BloodType":"A-",
"DrugAllergies":[
        "Aspirin","Penicillin"
        ],
# Embedding begins here
"Medication":[{
        "_id":"Med123"
        "Name":"Med1",
        "StartDt":"15/02/2014",
        "EndDt":"17/04/2014",
        "Freq":"2 Times a Day",
        "Dose":"10mg",
        "SideEffect":"Nausea",
        "PrescribedBy":"SCL054"
        },
        {
        "_id":"Med456"
        "Name":"Med2",
        "StartDt":"22/04/2014",
        "Freq":"2 Times a Day",
        "Dose":"10mg",
        "PrescribedBy":"SCL130"
        }]
}
```

```
> db.Doctors.find().pretty()
{
"_id":"SCL130",
"Name":"D1",
"Gender":"M",
"DOB":"15/08/1971",
"Speciality":{
        "Type":"Surgeon",
        "Area":"Thoracic"
        },
"Education":"M.D.(Thoracic)",
"Certifications":[
        "Surgery",
        "Thoracic and Cardiac Surgery
            "
        ],
# Referencing begins here
"PatientIds":[
        "OCLS1326",
        "DCMN9101",
        "LMNH459"
        ]},
"BloodType":"O+",
"Location":{
        "Hospital":"Fortis",
        "City":"Mohali",
        "State":"Punjab",
        "PinCode":"160162"
        }
}
```

Figure 3.8: Embedding and Referencing in MongoDB

Documents are addressed in the database via a unique key called *object_id* that represents the document. Both MongoDB and relation databases consist of *databases*

so, they can be compared. Inside a database, relational databases contain *tables* and MongoDB contains *collections*. Each *row* of a table corresponds to one *document* in MongoDB. The term *index* remains same in both. Corresponding to joins in relational databases, MongoDB provides two options — Referencing and Embedding. As shown in Figure 3.8, for referencing another document in an document, object_id of that document is used for referencing. Whereas, in embedding approach documents are nested such that all the information is present at one place only thus eliminating the need for referencing. Evidently, embedding approach incurs data duplication.

```
import pymongo
from pymongo import MongoClient
client = MongoClient('localhost', 27017)
db=client.HIS
patients=db.Patients
# Patients is the Collection
db.Patients.insert({'_ID': 'pat201406', 'Name': 'John', 'Age': 23, '
    Contact': '5199918289'})
# Inserted one document containg one patient details on collection
    Patient
list(db.Patients.find())
# For listing all contents of Collection Patient
db.Patients.ensure_index('Name')
# Indexing patient names
db.Patients.remove({'Age': {'$gt': 18}})
# Remove documents of all patients having age greater than 18
print db.Patients.count()
# Prints number of documents in Patients collection
```

Figure 3.9: Interacting with MongoDB in Python

MongoDB provides searching based on regular expressions on fields and also have provision of performing range queries. As shown in Figure 3.9, MongoDB allows indexing on any field, secondary indices are also possible. MongoDB provides high availability with the help of replica sets. Horizontal scaling is enabled using *sharding*. Based on a shard key chosen by user, data is distributed in a collection. MongoDB can be very well used as a file system, because of its automatic load balancing and data replication mechanism. Document-oriented databases are most appropriate for web-applications which require storage of semi-structured data and execution of dynamic queries. Built-in support for Map-Reduce enables aggregation of large amounts of data.

3.4 Graph Data-store

Graph database models are applied in areas where information about data interconnectivity or topology is more important, or at least as important, as the data itself. Graph databases are the best way to represent and query connected data, especially when size of data is significant [52]. With the mass scale successes of companies like Google, Facebook, Twitter which employed properietary graph technologies for their business models, graph databases have gained attention. These databases are tremendously suitable to model real world data which is highly interrelated. Graph databases store information that more closely resemble the ways humans think about data. In graph database modeling, data structures for schema and instances are modeled as graphs. A detailed survey of various graph database modeling is presented in [53], concentrating on data structures, query languages, and integrity constraints.

A graph database can traverse thousands of edges at a fraction of the cost of the relational joins because relationships are direct links between nodes [54]. Figure 7.3 shows the three basic components that build and annotates a graph are vertex, edge and property. Each vertex/node corresponds to entities in relational databases, where an edge connects nodes to represent relationship between them. Edges are labeled and directed where label identifies the relationship name and direction adds

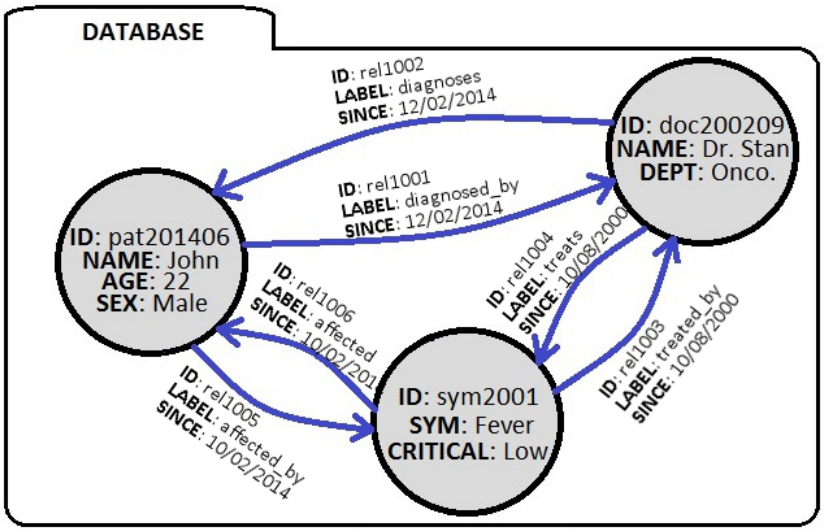

Figure 3.10: Data storage in Graph-based data-store

meaning to the relationship.

In today's era, we are surrounded with graphs like semantic web, natural sciences, social networks etc. Graph databases allows O(1) access to adjacent vertices because every element has a direct pointer to its adjacent element. Unlike relational databases, performance of graph databases do not degrade as size of database increases since the cost of local step remains the same because each vertex serves as mini-index of its adjacent elements [55]. Queries are performed using graph traversal techniques. Meaningful patterns emerge when one examines the connections and interconnections of nodes, properties and edges. Graph based data store is the backbone for all social websites.

Due to need of user-driven data it has become difficult to know beforehand what type of attributes will be needed. Relational databases work on pre-defined schema and there is no provision for dynamic and ad-hoc data. Almost all the available graph databases have capability of storing semi-structured information. It does not require a pre-defined schema which leads to easier adaptation to schema evolution and ability to capture ad-hoc relationships.

More than 20 graph databases are available of which few are proprietary and oth-

ers open-source, popular ones are Neo4j [56] and [54]. For these graph databases various query languages are available [57], Cypher for Neo4j [58], Gremlin which works for various graph databases [59], SPARQL query language [60] is used to retrieve and manipulate data stored in Resource Description Framework (RDF) format.

Use-cases: Pattern recognition, Recommendation systems, Navigational systems, Social networking applications, Spatial data.

Pros: ACID-complaint, Suitable for handling highly interconnected large amount of data.

Cons: Not good at horizontal scaling especially when related nodes are distributed across machines.

3.4.1 Neo4j

Neo4j is a powerful graph database written in Java. It has capability of storing billions of nodes and relationships efficiently and also provides very fast querying and traversal. It is available under both open source and commercial license. In addition to Java, Neo4j has bindings available in other languages too like Python, Jython, Ruby and Clojure. It also provides transactional capabilities and traverses depths of 1000 levels and beyond at millisecond speed [56].

Graph databases contain deeply connected data. To query this data which if stored in relational schema will result in joins of high depth. Each join will result in cartesian product of all potential combinations of rows and then filters out those that do not match the condition specified in the where clause. For one million users, the cartesian product of 5 joins will result in huge number of rows. Filtering out all those records that do not match the query will be very expensive. However Neo4j visits only those nodes that are relevant to the traversal so it's performance is not affected much with increase in data. Though, traversal get slower with increase in the depth because of the increased number of results that are returned.

```
from py2neo import neo4j
from py2neo import Node, Relationship
graph_db = neo4j.GraphDatabaseService("http://localhost:7474/db/data/")
Patient1 = Node("Patient", ID="pat201406", NAME="John", AGE=22, SEX="
    Male")
Symptom1 = Node("Symptom", ID="sym2001", SYM="Fever", CRTICAL="Low")
Doctor1 = Node("Doctor", ID="doc200209", NAME="Dr. Stan", DEPT="Onco")
diagnosed_by = Relationship(Patient1, "diagnosed_by", Doctor1)
treats = Relationship(Doctor1, "treats", Symptom1)
affected_by = Relationship(Patient1, "affected_by", Symptom)
graph_db.create(alice_knows_bob)
query = neo4j.CypherQuery(graph_db,"match (a) RETURN a")
print(a)
for result in a:
    print type(result) # py2neo.util.Record object
    print type(result.p) # p is a py2neo.neo4j.Path object
```

Figure 3.11: Interacting with Neo4j in Python

Neo4j can be queried through its native Java APIs, SPARQL, Gremlin or using Cypher query language. Cypher query language is used the most, especially for Neo4j graph database. Neo4j can also be queried via its shell. Visual editor for graph databases are also available, namely Neoclipse [9] and Gephi [61]. Neo4j Web Administration is the primary user interface for Neo4j, which is available at http://127.0.0.1:7474/ after installation of Neo4j server. Database backup facilities are also provided in Neo4j enterprise edition [10]. Importing data to Neo4j from MS Excel format is also possible using Py2neo [11] which parses csv (comma separated values) file and load it into Neo4j database. An example code snippet showing few basic operations of

[9] https://github.com/neo4j/neoclipse
[10] Neo4j Manual on Backup http://docs.neo4j.org/chunked/milestone/operations-backup.html
[11] Py2neo homepage http://py2neo.org

Neo4j using Py2neo is shown in Figure 3.11.

3.5 Conclusion

Each class of NoSQL database is suitable for different use cases. For example, if requirement is to simply store and retrieve non-transparent data items using key, key-value data-store like Redis is preferred. If application needs to store records with lots of attributes, but generally asks for few certain fields and also if the analytical functions are to be performed, column-oriented databases like HBase is recommended. Whereas, if the value associated with the key is needed to be searched and updated based on individual attributes within the value, a document database like MongoDB will be ideal. Storage of friend-of-a-friend type of data will be more natural in graph database as compared to any relational database, which demands complex joins of multiple tables. Although there are no fixed rules for mapping system requirements to each class of NoSQL data-stores, above examples are just a subset of features and their mappings.

Specialized representation of each data-store is accompanied with specific query languages. Each NoSQL data-store comes with its own query language and API bindings with various programming languages like Java, Ruby, Python etc. Learning query language for each NoSQL data-store is an additional overhead, but at the same time provides flexibility in querying underlying data in an expressive way. Absence of a uniform query language to query NoSQL data-stores is one of the biggest hurdles in quick adoption of NoSQL data-stores. Efforts have been made in this direction, but are not very successful[12], due to diverse nature of these data-stores, their querying capabilities and query languages.

[12]UnQL (http://unql.sqlite.org/)

Chapter 4

Data Integration Techniques

Data integration deals with providing a uniform and transparent view of data stored in multiple, heterogeneous, and autonomous data sources. Various projects have been developed since late 1970's for integrating heterogeneous databases. Initially, focus was primarily on integration of relational databases with legacy data storage systems. Subsequently, with emergence of object-oriented and XML databases; efforts were made in the direction of integrating them with relational databases.

Popularity of new class of databases - NoSQL and NewSQL, necessitate the need of developing application that partially make use of new databases, while keeping some data in mature RDBMS; hence there is an urgent need for inter-operation among these databases.

Researchers have designed different approaches and mechanisms for achieving data integration, employing mechanisms like data-warehousing, mediated schemas, data-dictionary, knowledge-bases etc., some of them are explained briefly in this chapter. This chapter also discusses systems which combines relational and non-relational databases.

4.1 Data Integration

Purpose of data integration is to provide a mechanism for binding a group of databases together [62]. Challenging aspect of data integration process is that the involved databases are heterogeneous in nature, having different syntax and semantics and exposing different capabilities. To integrate them as a single unit, pre-processing at different levels are required. Data integration process consists of two important activities – constructing a unified query interface and maintenance of integration system. Maintenance activity encompasses various tasks related to efficient execution of queries and handling of change in schemas of used data-stores.

Value of data is increased manifold when disparate data-sets are brought together. It also increases the availability of data, allowing anyone to retrieve, examine and analyze the consolidated data. Also, with the availability of wide range of integration tools, expectations of users have also increased substantially, users are no more limited to performing simple queries but also performing complex and bulk queries over disparate data sources. Various issues related to data integration includes expressiveness of common data model, expressing semantic correspondences across involved data sources, resolving semantic conflicts between data sources prior to integration, query decomposition and translation and optimization [63].

Major obstacle in achieving interoperability among disparate systems is data and system heterogeneity [64]. *Data heterogeneity* refers to the different ways of organizing and interpreting data of various data storage systems. *System heterogeneity* refers to differences in data modeling, data definition and manipulation languages, concurrency control mechanisms etc. Data heterogeneity can be further divided into – Schematic, Semantic and Intensional conflicts [65]. Schematic conflicts means conflicts in data-types, data labeling, data aggregation and data generalization. Scaling and naming conflicts come under semantic conflicts. While, integrity constraints or domain conflicts are covered in intensional conflicts [66].

Process of integrating a number of data sources is called source integration, which is further classified into two classes – *schema integration* and *data integration*. In schema integration, schemas of underlying data sources are joined intelligently to ob-

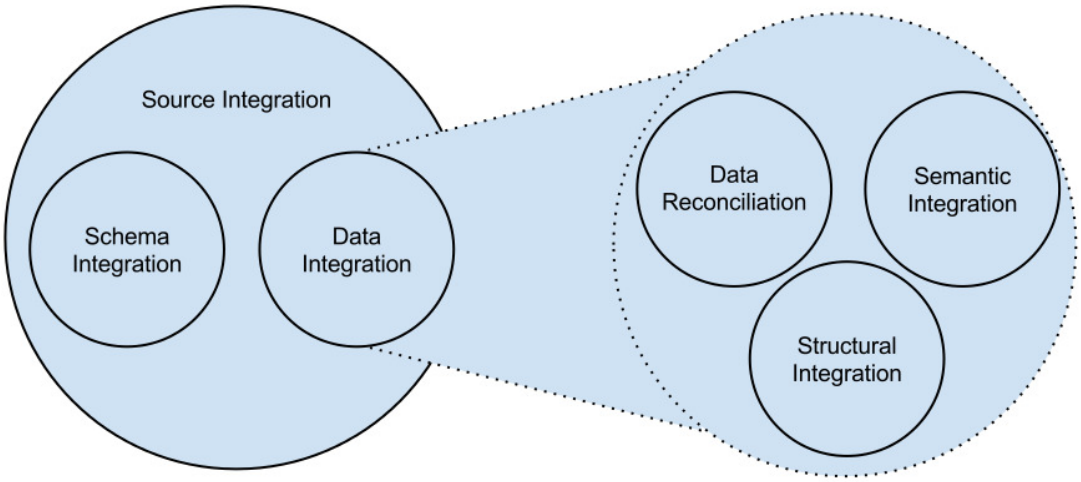

Figure 4.1: Types of data integration

tain a common integrated schema, against which queries are performed [67]. Data integration on the other hand, provides a data catalog which contains data assimilated from all the involved data sources [68]. Data integration is also called information integration and it provides add-ons over schema integration by not only integrating the schema, but integrating the underlying data as well. Undoubtedly, data integration approach violates the principles of data autonomy and data distribution. As shown in Figure 4.1, data integration is further decomposed in several sub-areas.

- *Structural Integration* deals with the problem of structural heterogeneity such as heterogeneities in data modeling techniques, querying mechanisms etc.
- *Semantic Integration* resolves semantic mismatches between involved schemas, which are result of differences in conceptualization of multiple database engineers. Semantic mismatches may occur at both data and schema level.
- *Data Reconciliation* encompasses data related problems such as *object identification* and *data cleaning*. Object identification determines the associativity of objects belonging to heterogeneous data-sources. Data cleaning on the other

hand take care of the errors encountered during data acquisition. Data cleaning problems are further categorized as – invalid data and representational differences of data.

Lazy Vs Eager Integration

Lazy integration can also be understood as **on-demand integration**, that is data from underlying sources is integrated only when the user asks for it through queries. And only those sources will be integrated which are specifically asked by the user. During other times, data remain at rest in their respective data sources. In contrast to reactive approach of lazy integration, eager integration follows an pro-active approach – data is extracted, cleansed and put at a central place beforehand. All the queries are then asked against this central data repository.

Answers may be stale in case of eager integration, while in lazy integration, user always gets up-to-date answers. Query processing is the foremost judging factor here, in eager integration query processing is done locally at central repository and hence performs faster and can also fetch results even when data sources are not available. On the other hand, in lazy integration query processing is slower as data is present in data sources itself and it may also interfere with the local query processing.

Materialization Vs View Integration

Whether materialized copy of integrated databases is at one place or whether integration describes the process of query translation where queries are asked against the view presented to the users, and are then translated into sub-queries of target data-stores. In materialized approach, a central repository is made a-priori, all queries are asked against this materialized database [69]. Whereas, in view integration sources are accessed on the fly that is when query is asked [70]. Materialization provides better performance, enables local query optimization, eliminates inter data-store communication and is more reliable. However, initial cost of setting up materialized integrated database is high and the maintenance is also costly [71].

Read-only Vs Read-Write Integration

Ideally, data integration systems should provide consolidated access to the data-sources and allow modifications (delete, update) also. But these modifications have to be propagated through the integration systems to the data-sources, which necessitates a *tight integration* of the underlying data-sources and the integration system [72]. Alternatively, *loose integration* provides read-only access to the sources via the integration system thereby maintaining autonomy of the data sources and resulting in a simpler system [73]. Generally, in read-only integration systems the data-sources and the integration system belong to different parties.

Data Translation Vs Query Translation

In data translation approach, first a global database which contains unified data in a common format, collected from all local data sources, is constructed [74]. All required data is present at same place, hence it becomes easy to execute queries. Data is duplicated in both local and global databases which leads to wastage in storage space. For each update performed in local data-stores, global database needs to be updated as well which causes significant overhead. It is difficult to combine data present in heterogeneous data-stores into a common format of global database keeping various constraints present in local data-stores intact. Any addition and removal of local data source causes a significant overhead to add/remove data from global database.

Under query translation strategy, a global virtual view is created considering all local data-stores, although data is present in local data stores only [75]. Global query asked with respect to global schema (virtual) is translated into local queries by a mediator. Results, which may be in different formats, are obtained after local query execution from local data-stores are combined. As compared to above explained approach, addition or removal of data-store is easier in this alternative.

Procedural Vs Declarative Mapping

Query translation strategy is further classified on the basis of mapping; procedural or declarative [76]. In *query translation based on procedural mapping*, the mapping between virtual global database schema and multiple underlying databases is done with the help of procedural functions which include objects from local data-stores and corresponding objects in the global schema. This approach has been used in systems like physician workstation at HP Labs[77], Kliesly system [78] and many more. This approach gives adverse performance if data-stores keep changing. Every change in data-store introduces changes in procedural functions also.

On the other hand, in *query translation based on declarative mapping*, a declarative representation specifies the correspondence between objects and operations of global and local schemas [79]. Query optimizer module will inspect these declarative mappings to formulate the query execution plan. Declarative mappings are easy to maintain and reflect changes in underlying data-stores as compared to procedural mappings. TransFER system use declarative mappings by using an extended version of relational algebra to define mapping among semantic data model and inherent relational databases [80].

4.1.1 Multidatabase and Federated Databases

Multidatabases provide integrated access to multiple relational databases, whereas in federated databases component data sources can be of any type; not necessarily relational databases [81]. In *multidatabases*, information regarding distribution of data sources is hidden from the user, but the system does not hide schemas of component data sources from the user. Table names are prefixed with database's name. Queries and updates are performed in two phases, one at local level, second at global level [82].

Limitations of multidatabases are –

1. User of the system is supposed to be fluent in SQL and should be aware of how data is distributed among databases.

2. Involved databases must be relational databases only.

3. Absence of metadata regarding component databases complicates the process of query processing and results unification.

Federated systems integrate underlying data sources with respect to both data model and schema, and the data is aggregated on demand basis [72]. These systems consist of three layers - *Integration*, *Export* and *Data*. Integration layer provides unified view of information present in data sources. Each database has its own export schema which defines how its data will be accessed and viewed. Data layer compromises of the component data sources. If the integrated schema is defined by the user, system is called loosely couple federated system, whereas if defined by administrator it is referred to as tightly coupled federated system.

Two main components of federated systems are federator at integration layer and wrappers at data layer [83]. Federator is responsible for query decomposition and distribution and finally aggregation of results. Wrappers support export schema by mapping the component data models represented by the integrated schema. Benefits of federated systems are support of ad-hoc queries, provision of fresh and current data and requirement of a very less additional storage. Disadvantages are low response time from data repositories and overhead due to translation between various source schemas and data models.

Mediation systems are very similar to federated systems except that the mediation-based integration is more light-weight and flexible. These systems primarily support read only queries. Mediator takes place of federator. Unlike federated systems which follows bottom-up approach, that is the data sources those need to be integrated are known beforehand; mediation systems are engineered in top-down fashion [84]. Mediation based integration is explained in more details in the next section.

4.1.2 Mediation based integration

Data integration systems enable users to query multiple data sources without worrying about actual location and format of the data source. Users are free from having

to locate sources relevant to a query, interact with each one of them in isolation, and then manually combine data from multiple sources to provide final result to the user [85]. Queries from user are no more directed to the schema of underlying data sources, instead are posed on mediated schema. Mediated schema gives a consolidated view of the disparate data sources to the user. Mediated schema can be constructed either manually or automatically [86], also its linkage with underlying data sources can be specified by mappings as GAV (Global-As-View) or LAV (Local-As-View).

Global/mediated schema is defined in terms of data sources in GAV approach, and local data sources are defined in terms of the global schema in LAV approach. Any change in local data sources do not affect global schema in LAV because local schemas are defined as views over the global schema, but query processing can be complex. On contrary, in GAV since every relation in global/mediated schema is mapped to a view over source local schema, querying is simple but for each change in local data source global schema needs to be updated. Detailed comparison of these two approaches is reported in [62, 73]. Another approach called GLAV (Global-Local-As-View) that combines the GAV and LAV views together is available in literature [87, 88]. The difference between GAV and LAV approach of mediation is explained with the help of an example.

Consider a relational global schema having following relations:

blogpost(Title, PostingTime, AuthorName)
technical(AuthorName)
comments(Title, Bloggers)

The source schema contains 3 source relations namely r_1, r_2, and r_3. r_1 stores the blog details since 2014, while r_2 stores comments posted by other technical bloggers on a post and r_3 stores name of all technical bloggers.

r_1*(Title, PostingTime, AuthorName)*
r_2*(Title, Bloggers)*

r_3*(AuthorName)*

The LAV mapping is given by the following views,

r_1(T, P, A) ← blogpost(T, P, A), (P ⩾ 2014)
r_2(T, B) ← blogpost(T, P, A), comments(T, B), technical(A)
r_3(A) ← technical(A)

Whereas, GAV mappings are given as follows:

blogpost(T, P, A)← r_1(T, P, A)
comments(T, B) ← r_2(T, B)
technical(A) ← r_3(A)

Popular implementations of LAV approaches are Information Manifold [89] and InfoMaster [90], and Systems those make use of GAV for integration are TSIMMIS [91], HERMES [92] and Garlic [93]. The TSIMMIS data-integration system provides integrated access via architecture that converts data from each source into a common object-oriented model. Wrappers (also called translators) provide a common query language and common data model for extracting information, and also translates asked query into source-specific query. Wrappers also translates the data received from each source into common data model.

4.1.3 Data Warehouse

Data warehousing is one of the most common eager integration method where data from underlying data-sources is integrated in advance into a centralized repository called as *data warehouse*. Intended user of the system execute queries against this data warehouse. Data warehouses are tightly integrated and provide fully materialized solution. It must be noted that since the data is combined from multiple data-sources, data must be first filtered and translated to be able to finally merge

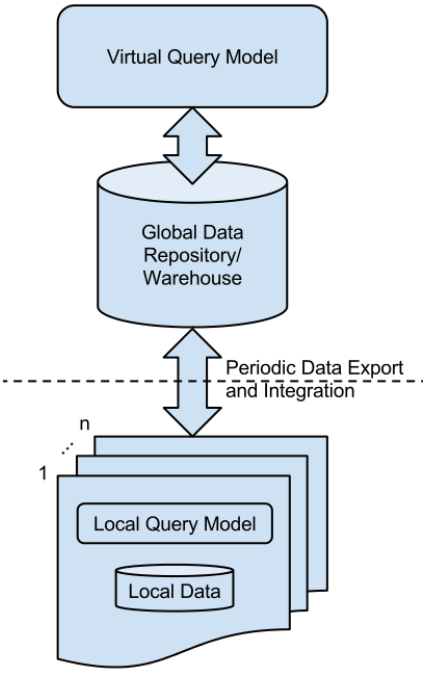

Figure 4.2: Data Warehouse Architecture

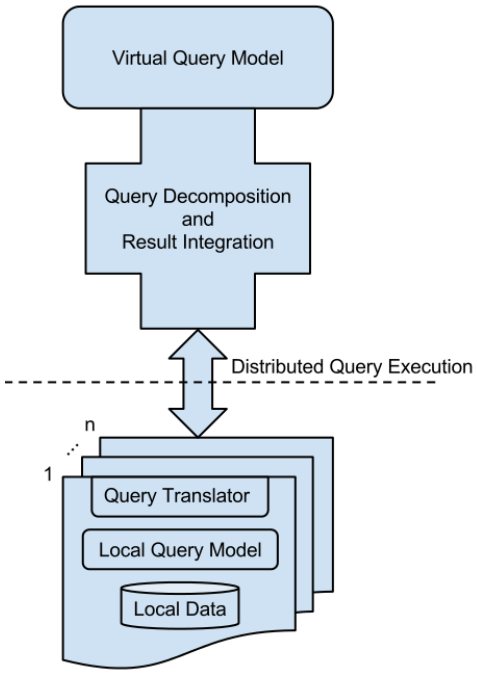

Figure 4.3: Mediation Architecture

into a common database schema, this process is also widely known as *ETL (Extraction Transformation Loading)* process [94]. Visibly, data warehousing follows data translation approach in contrast to query translation approach (followed in mediation architectures).

Data warehousing approach towards integration is appropriate for scenarios where specific predictable chunk of data is asked by the client. Also, where the user does not require current data, but performance of query is given priority. Major research issues in implementation of warehouse approach is accumulation of data from underlying heterogeneous data sources, which generally are present in multiple formats, another important issue is incorporating changes in underlying data sources in the central data warehouse at regular intervals [95].

Advantages – Advantages of data warehousing approach are largely derived from the usage of central repository, thereby avoiding problems which are generally encountered in distributed systems such as network bottlenecks, unavailability of data

sources low response time etc. Another important advantage is the absence of inconsistencies related to data representation because data from multiple heterogeneous data sources is translated into a common format before loading.

Disadvantages – Efforts made during the process of translating and merging the data from multiple repositories to a common database are very high and should not at all be repeated frequently. Large amount of overhead incurred during the setting up of data warehouse makes it very difficult to perform any update of data as well as addition or removal of repository. Data present in data warehouse is mostly stale, until repository is again updated with involved data sources.

Architecture of a data warehouse is shown in Figure 4.2, which shows that data from involved data sources is periodically exported into the global data repository also known as warehouse, which is queried by the user. Contrasting it with mediation based integration architecture shown in Figure 4.3, data is not exported instead the query itself is decomposed and then sent to the underlying data sources, after execution the results are integrated at the mediation layer and presented to the user. Evidently, in data warehousing approach, efforts are made during exporting and warehouse creation whereas in mediation approach, efforts are made every time a query is asked.

4.1.4 Ontology based integration

Ontologies are defined as "explicit specification of a conceptualization" [96], thus ontologies can be used for stating the semantics of data sources for data integration purposes. Ontologies can also be used for identifying and associating semantically correlating concepts from data sources. Ontology facilitates integration process by providing vocabulary that represents the semantic terms and relationships of a domain. In ontology based data integration, objective is to provide access to data stored in heterogeneous databases through semantic layer formed using the concept of ontology. Relationships between entities in ontology and data in databases is stated using semantic mappings, as shown in Figure 4.4. In addition to explicating the contents of data sources, ontologies have also been used as global query schema

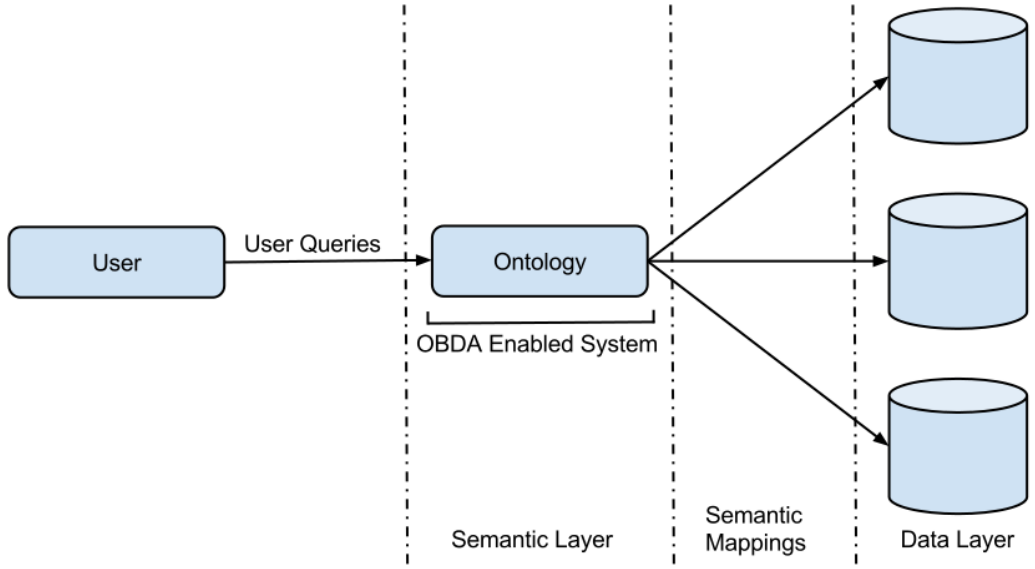

Figure 4.4: Ontology-based Data Integration Architecture

[97] which provides a more intuitive environment to the user of the system. Ontologies also plays a very vital role in automatic verification of schema mappings between local and global schema and vice versa [98].

Ontology based integration architecture is classified into three groups – *Single Ontology*, where ontology acts as global reference model in the system [99], *Multiple Ontologies*, where each data sources is modeled in an ontology are combined which require mappings among ontologies also in addition to mappings between ontology and data sources [100], and *Hybrid* approach where multiple ontologies are used, but all of them subscribe to one common vocabulary. Usage of ontology for data integration has its *advantages* like – providing a common vocabulary that acts as a steady theoretical interface, being independent of the underlying database schemas, language offered by ontology is expressive enough for handling complex queries, and ontology represents knowledge in sufficient detail so that description of all relevant data sources can be merged into one common frame of reference.

Another type of integration is *Link based Integration*, where links are used to integrate data present in different data sources. This approach is based on point

and click mechanism and thus the path of links followed make up the information pertaining to a query [101]. Architecture of link-based integration software is very simple and physical integration of data is not required. *Advantage* of link based integration is its simplicity, model does not demand expertise from the user of the system. Setting up the system and then updating it only requires creation and updation of an index. Its disadvantage is also due to its simplicity, since users are restricted to search only keywords and have to browse multiple web pages manually to access any required information. Heavy dependence on index is also a disadvantage because if the initial design of the index is not correct, then possibility of overlooking of crucial information may occur.

Each of the data integration approaches explain above have its own advantages and disadvantages. Multi-database approach of data integration has inadequate location transparency, requiring the end user to be aware about the location of required data. Federated systems demand that the data sources to be integrated must be known before-hand as it follows bottom-up approach towards integration. Data warehouse incurs high overhead costs associated with creation and maintenance of centralized repository. Ontologies enable semantic integration of heterogeneous databases with the help of shared vocabulary. All of these integration techniques were extensively studied using relational, object-oriented and XML databases. Integration of modern NoSQL data-stores with existing databases is still in its inception.

4.2 Polyglot-Persistence

Data integration follows TOP-DOWN approach, while ***polyglot persistence*** follows BOTTOM-UP approach. In data integration, independent data-sources already exist functioning autonomously, only when the need to provide an unified view of various underlying sources emerge, a layer of data integration is incorporated in the framework. Whereas in polyglot persistence, the framework is designed from scratch where the data storage layer comprises of more than one type of data-store. It is decided at the starting itself that data of the application will be stored in more than one

data-store.

Modern applications need to store and manipulate data in multiple data-stores repeatedly. However, interacting with heterogeneous data-stores is not an easy task when it comes to managing complex queries such as joins. To make matters worse, each NoSQL data-store exposes its query language and data model. To rule out these problems, researchers have proposed several solutions to provide transparent access to heterogeneous data-stores. Some of the proposed solutions are based on the definition of a common API while others are based on frameworks able to access different data-stores.

Also, with the availability of a plethora of NoSQL databases, developers are now equipped with a range of databases to choose from before starting their application. Developer may also wish to use more than one data-store in an application depending upon the specific requirements of the application. An application that makes use of more than one data-store is called *polyglot persistent application* [1]. Polyglot persistence has already been well adopted by organizations like Facebook, Twitter, YouTube and Pinterest. In polyglot persistent solution, data is distributed across multiple data-stores therefore, there is no single point of failure as is the case with an application using centralized RDBMS. Each available data-store comes with its own strengths and weaknesses, first step towards achieving polyglot persistent solution is to select data-stores those meet the application's requirements completely.

Table 4.1 gives comparison of existing solutions which deal with multiple classes of NoSQL data-stores along with supported data-stores, their querying mechanism, and implementation approaches. SOS [102] and ONDM [103] allow application to query data from different NoSQL data-stores but do not support processing of complex queries which requests data from multiple data-stores. SOS provides CRUD operations at the level of individual data-store. These operations are provided via GET, PUT, and DELETE methods. Interoperability among different classes of NoSQL data-stores is transparent and is provided by accessing them via a common interface. Metamodelling approach has been used for mapping data storage and querying mechanism of involved data-stores with the common model.

ONDM provides ORM like API based on popular Java Persistence API (JPA) to

Table 4.1: Detailed comparison chart of existing polyglot-persistent solutions

Integration System	**Supported Data-stores**		**Querying Mechanism**	**Approach**
SOS [102]	Column:	HBase	PUT, GET, DELETE	*Metamodelling*: Handlers of each data-store mapped to common Meta-Layer
	Document:	MongoDB		
	Key-Value:	Redis		
ONDM [103]	Column:	Cassandra	Object-Oriented API, based on JPA	*NoAM* (NoSQL Abstract Model) [104]
	Document:	MongoDB		
	Key-Value:	Redis		
Curé et al. [105]	Column:	Cassandra	Bridge Query Language (BQL)	*GAV*: Mediated (Relational) Schema and set of mapping assertions
	Document:	MongoDB		
	Relational data-store			
Curé et al. [106]	Column:	Cassandra	SPARQL	*DL* (Description Logic)-based *OBDA* (Ontology Based Data Access)
	Document:	MongoDB		
	Relational data-store			
Roijackers et al. [107]	Relational:	PostgreSQL	NoSQL Query Pattern (NQP)	Hybrid read-only data-store combining relational and non-relational data
	Document:	MongoDB		
Unity [108]	Relational:	MySQL	SQL)	SQL to NoSQL translator, Data Virtualization
	Document:	MongoDB		
ODBAPI [109]	Key-Value:	Riak	REST API	Unified REST API for performing CRUD operations
	Document:	CouchDB		
	Relational data-store			
CDPort [110]	Column:	Amazon SimpleDB	Common API	Platform independent data abstraction layer using adapters
	Document:	MongoDB		
	Key-Value:	Google Datastore		
Proposed Framework - **PolyglotHIS**	Relational:	PostgreSQL	Native Query Language of data-stores	Mediating Layer, Datalog for KBoD and JSON for result unification
	Document:	MongoDB		
	Graph:	Neo4j		

application developers to interact with NoSQL data-stores. ONDM approach capitalizes on common features of NoSQL databases by representing data as collection of aggregates, few initial activities are thus autonomous of the target data stores. This approach provides scalability, consistency and performance with the help of its basic data model – *aggregates*. In line with JPA, it is heavily influenced by entity data model, consisting of entities, relationships, and complex objects.

Curé et al. [105] also attempted to combine three classes of data-bases, namely relational, document and columnar. They represented the global schema by a relational data model to query NoSQL and relational DBMSs and defined a mapping language to map attributes of the data sources to the global schema and a Bridge Query Language to rewrite queries. BQL acts as a bridge between specific query language of the involved data stores and the SQL query language which the user uses for expressing the query. Preferred access paths are stored for each mapping assertions, which also deal with data conflicts. Instead of data translation, query translation approach is employed. Data remains at data sources itself, one centralized virtual relational database is created against which queries are asked in SQL.

In a second step, Curé et al. [106] extended their solution by using Ontology Based Data Access (OBDA). Additionally, they replaced BQL with SPARQL[1]. Although their proposal is promising, it lacks in some functionalities viz. no query optimization at global level and no complex query execution.

Roijackers et al. [107] proposed a hybrid mediation approach in which they integrate relational and document data-stores. More precisely, documents are logically embedded inside relational data-stores to form hybrid data-stores queried via the proposed extended-SQL language called NoSQL Query Pattern (NQP). Reasons for loading NoSQL into SQL and not vice versa were the maturity and advanced capabilities of SQL. Unity [108] project is another attempt at unifying relational and document-oriented NoSQL databases which provides an SQL interface for querying. Unity automatically translates SQL queries into data store specific query syntax. Unity virtualization system architecture consists of SQL Query parser, Query translator, Query optimizer and virtualization execution engine.

[1]http://www.w3.org/TR/rdf-sparql-query/

ODBAPI [109] provides a unified REST API for combining relational and two classes (key-value and document) of NoSQL data-stores. Interacting with multiple data-stores demands knowledge of API of each of the target data-store. ODBAPI alleviates this burden by enabling CRUD operations on underlying data-store with the help of one REST API.

CDPort [110] provides an extensible framework enabling data portability as well as data exportability among cloud-based NoSQL data stores. A common data model and API is implemented which eliminates the need to change software code if underlying cloud-based storage model is changed. CDPort also significantly reduces the work demanded for migrating data from one storage solution to another, especially within different classes of data stores. The tool is extensible to support other than the already supported three data-stores - Amazon's SimpleDB, Google Datastore and MongoDB by creating new adapters for the target data-store. Other similar projects which deal with the issue of interoperability in cloud storage systems are moSAIC [111], Cloud4SOA [112] and MODACLOUDS [113].

We have developed an intelligent information integration solution–**Polyglot Healthcare Information System (PolyglotHIS)** which uses multiple data-stores [114]. The rationale is to select the most appropriate data storage technology that meets the specific requirements of each module of HIS. Our approach is novel in the way various data-stores are integrated. Apart from NoSQL data-stores, multiple co-operative agents are also employed. The system also makes use of Datalog for integrating these data-stores. Datalog which is a declarative logic programming language used to store set of facts and rules, helps in the storing and inferring of the capabilities of data-stores used in the PolyglotHIS.

Biggest hurdle in the integration of NoSQL data stores is the support of different query languages for each data store. Attempts have been made by the researchers to develop a common query language which can query as many NoSQL data-stores as possible. Valer et al. [115] proposed a solution that maps data from various NoSQL models to XQuery and XPath Data Model (*XDM*) which can further be queried using XQuery. UnQL [116] is a superset of SQL and supports JSON document structure containing collections and documents. Another attempt towards standardizing query

language for NoSQL databases is JSONiq [117], which is a XQuery based language for JSON. JSONiq is superset of XQuery and primarily supports MongoDB and CouchDB.

In [118] Debashish Ghosh proposed an multi-paradigm database architecture that uses multiple NoSQL data stores for different modules in an application to store the data closer to its representation and usage. In addition to usage of multiple frontal data stores, proposed solution used relational database as centralized underlying storage for generating reports, execution of batch processes and other work. Relational database is synchronized with other non-relational data stores with the help of asynchronous message passing, which being asynchronous leads to attainment of eventual consistency. Asynchronous updates does not limit scalability and availability. But, Eventually consistent solution is not acceptable to everyone. For example banking applications requires data to be ever-consistent.

Medical database integration will lead to overall development and well being of human in addition to benefits provided to medical/research organizations. Integration of bio-informatics will serve as back-bone to research of 21st century's life sciences. Integration of the medical data is vital for improvement of patient care, biomedical research, working of medical organizations and public health overall [62].

Heterogeneous database integration in context of hospital information systems (HIS) have been extensively studied in literature [119]. Hitherto work in using multiple data-stores in HIS contemplated integration amongst multiple relational data sources and later on also considered object-oriented and XML data sources. Each new technology up-brings new prospects of improving the existing work. Analogously, NoSQL data-stores have also clinched popularity in coherent storage of health-related data. Use of individual NoSQL data-stores for implementation of specific use-cases have already been studied in the literature.

The performance of NoSQL, XML-enabled and native XML databases for storage of structured clinical data have already been compared[2]. NoSQL data-store was discerned advantageous in terms of query execution time, while XML databases provided better flexibility and extensibility. XML databases have been proven to store EHRs better than relational databases [120]. In [2] authors compared performance of

NoSQL, XML-enabled and native XML databases for storage of structured clinical data.

MongoDB have been used for storage of ethnomedicinal data available in multiple sources [121]. Support for flexible schema in MongoDB empowered incorporation of both standard and customized ethnomedicinal plant data format. Another NoSQL data-store, columnar database *HBase* exhibited proficiency for storage of EHRs using distributed storage model [122]. Hybrid approaches have also been advocated, e.g. the use of hybrid row-column two-level database for storage of DICOM files which stores regular information about patients, as well as images and videos have been recommended [123].

Jiang et al. [124] have implemented an polyglot-persistent framework involving both structured and unstructured data to manage voluminous and diverse nature of data generated by sensors and RFID readers. Data was distributed among file repository, MySQL and MongoDB. In another work, *pyEHR* is a toolkit for developing scalable health-care data management system using MongoDB and ElasticSearch [125].

Integrating data from disparate clinical data sources is an enduring research topic. Emergence of NoSQL databases have generated much excitement and interest in the bio-informatics research community. Concept of polyglot persistent application is still in its inception and to the best of our knowledge have not been implemented in context of healthcare information systems. Although, few systems have been proposed recently that store data in (and query from) multiple data-stores. Stonebraker [126] presents the problems and the limits that a user may encounter while using NoSQL data-stores. These problems derive from the lack of standardization and the absence of a unique query language and API, not only between NoSQL data-stores but also between relational data-stores and NoSQL data-stores.

4.3 Conclusion

Data integration deals with combining data present in multiple homogeneous or heterogeneous data sources, enabling user to query underlying data-sources from a unified view. Two popular approaches for achieving this are data-warehousing where data from all the sources is transformed into a common format and physically loaded into a single database, and mediated schema where data resides in the data-sources while user can pose queries against a Global Virtual View (GVV) which provides a integrated, virtual view of the underlying sources. Data-warehousing approach incurs overhead during assimilation of data into a uniform format, especially while dealing with heterogeneous data-sources. Addition or removal of data-sources result in updation of complete data-warehouse, again an intimidating task. Query processing is however very efficient since all the data is present at one place and in an uniform format.

Query processing in mediated schema involves various steps starting from determining the required data-sources, query decomposition into multiple sub-queries, sub-queries execution at underlying data-stores, and finally result unification. Other components worth mentioning are - Query planning and query optimization. Addition or removal of data source is easy and data present in autonomous data-sources in multiple formats need not to be converted into a uniform format. Data is queried in-place.

In this chapter, we discussed various data-integration techniques accompanied with advantages and disadvantage of each of them. We also discussed the polyglot-persistence approach towards application development where more than one type of database is employed for data-storage purposes. The essence of this approach is to use the best data-base according to the storage and query requirements of each module, which may lead to use of many databases within an application. The advent of NoSQL databases have undoubtedly increase the options available to the developer community to chose from. In next chapter, we have presented the architecture and components of our system – *PolyglotHIS* which makes use of three different data-stores.

Chapter 5

Proposed Multi-paradigm Framework: PolyglotHIS

Persistence needs of applications are shifting from mostly relational to a mixed data-stores paradigm. Various modules within an application should preferably use different data-stores to model data closer to its semantic usage. Hence nowadays use of hybrid data-stores is emerging as a viable solution.

Amalgamation of different databases within an application is known as Polyglot-persistence. Pre-requisite for achieving polyglot- persistent solution is the availability of different types of databases. As late as 2005, relational databases ruled as the de-facto databases but now their reign is challenged by the advent of non-relational databases known as NoSQL data-stores, making Polyglot Persistence possible.

This chapter presents the detailed architecture of proposed framework – PolyglotHIS which relieves the end-users from interfacing with each involved data-store individually. We have considered Healthcare Information System (HIS) as a case-study, as it deals with variety of data corresponding to various departments in the hospitals. Thus justifying the need of polyglot persistent solution.

5.1 Introduction

Any practical multi-platform software assimilating data of different formats and horizons demands different querying modes. A software which make use of more than one type of data-store is called *polyglot persistent* software. This term was proposed by Pramod Salvage and Martin Fowler [1] and takes its origin from the term *polyglot programming*, which is refers to a software which uses more than one programming language. Overhead of polyglot-persistent solution lies in execution of queries that require data from more than one data-store to give complete result, because there is an impedance mismatch between various data stores in terms of data structures, granularities, query languages, backup and restore strategies [127].

By choosing appropriate data-store and using its benefits to store data which it handles best, overall improvement in performance can be achieved [128]. To get data from different data sources, one needs to first learn different user interfaces and then cross-reference these data sources. Data sources contain lot of useful information, but due to unavailability of uniform/integrated interface to query across them, lots of questions remain unanswered. Following steps to used to implement a polyglot persistent solution

1. Identify use-case and analyze the requirements : *Is polyglot persistent solution really required?*
2. Literature survey of existing database solutions : *Which databases to chose to store the data in hand?*
3. Architecture and integration approach : *Which integration approach to use?*

First the requirements of software are analyzed thoroughly and if the application deals with variegated data only then an implementation of a polyglot-persistent solution is planned. Planning is required because polyglot-persistence comes with a cost and hence must be adopted only if utmost necessary. Once decided to implement the polyglot persistence the most important step is to choose which databases to use for various modules within an application. This decision is getting tougher day by day with the availability of large number of choices in data storage solutions [129].

Various research papers and blog posts have given very well observed comparison of well known databases [130–134]. They must be studied carefully and specific requirements of each module are then mapped to different data-stores. Chapter 3 has presented a detailed study of all four classes of NoSQL data-stores together with one popular data-store implementation of each class. We have also presented how each of the data-store that can be queried and manipulated using Python programming language. The final step that is selection of the integration strategy, has been discussed in Chapter 4 by explaining various possible integration solutions.

Polyglot-persistent softwares are gaining popularity and are being adopted for implementation in various fields [135]. Researchers have proven the success of polyglot-persistent software development methodology over traditional method of using one database only in context of health-care [136], energy data management system [137], energy consumption control [138], LAIS (Library, Archival and Information Services) [139], digital forensics [140], biometrics domain [141] and geo-spatial big data management [142].

Functionality of the implemented polyglot-persistent framework – Polyglot Healthcare Information Systems (PolyglotHIS) can be described with the help of following steps: (a) end-user (doctor/nurse) submits a query to the integrated system, which may require data from more than one underlying data-stores; (b) mediation layer composed of various co-operating agents processes the user request and determines how to split the asked query into sub-queries each pertaining to a different data-store; (c) sub-queries are executed on the underlying data-stores and individual results are returned to the reducer agent; and (d) results are merged and shown to the user in a desirable format by the graphical interaction agent.

5.1.1 Polyglot-persistence in Healthcare Data Management

With the extinction of paper-based record keeping hospitals, data management is no more an optional component of healthcare systems, rather it is now regarded as spinal chord of these systems. One visit of a patient to a healthcare organization or a stay at hospital generates thousands of data elements like new clinical record, diagnosis,

medications, lab results, billing and so on. In 2012, worldwide estimated digital healthcare data was around 50 petabytes and is expected to reach 2,500 petabytes in 2020 [143]. This *Voluminous* data is accompanied with *Velocity* (real-time data streaming) and *Variety* (semi-structured EHRs). Markedly presence of 3 V's of big data in healthcare has marshalled the notion of *healthcare big data* [144]. Like an e-commerce website that predict *'you might also like'* products for its customers, clinical institutions can also tap healthcare big data to provide better facilities to their patients.

In unison, it is absolutely necessary to store the clinical data in its entirety to be able to later deduce useful information from it. Storage of health-care data as a whole is vital for patient himself as well as for other patients of same age and gender or having same symptoms or diagnosis for reference. However one should be careful that the system must be adopted as a whole and not on a piecemeal basis as the absence of even a small piece of information could lead to wrong results and could prove disastrous for the patients and the reputation of the hospital. For example, if information about smoking habit of a patient diagnosed with kidney failure is not stored in the system, the system may not infer the cause and effect relationship. This can lead to misleading results which can play havoc with the life of the patient and in-turn give bad name to the hospital.

Researchers have proved that non-relational databases are more suitable for modeling and storing clinical data present in Electronic Health Record (EHR) in comparison to relational databases [122, 145, 146]. An EHR is a patient record created by organizations like hospitals and clinical centers. EHR stores basic information as well as complete profile of a patient including medical history of patient's family and patient's life style. EHR also stores symptoms, diagnosis and suggested/ongoing treatment for each visit of the patient [147]. Relational databases are good at storing structured data, whereas EHR is predominantly semi-structured. Querying capabilities of a health-care information system includes interactive queries that provide assistance to the doctors in reaching to a conclusion in the form of treatment, after analyzing patient's symptoms and profile. We can also look out for treatment effectiveness by reviewing EHRs of a large number of other patients [148].

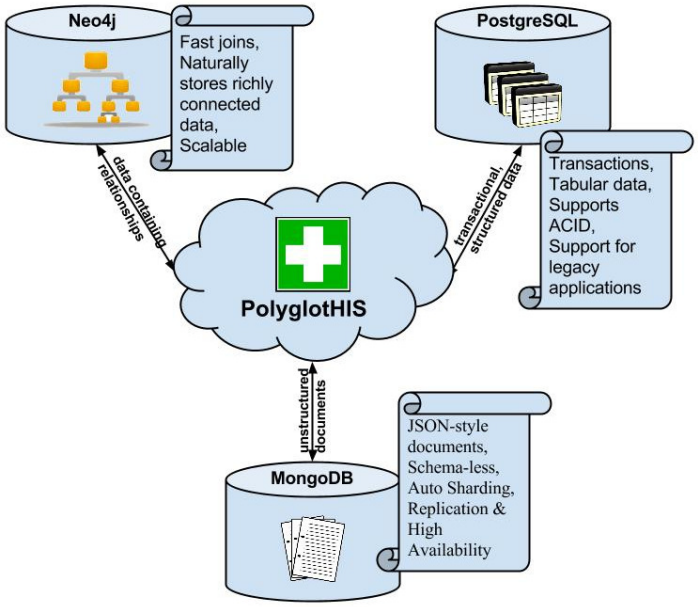

Figure 5.1: Architectural Overview of PolygotHIS (Proposed System)

Health-care information systems support interactive queries as well as analytical queries. They also embody data about symptoms, diagnosis, various possible treatments, medicine drug composition, doctor's specialty, besides data about various procedures and activities of a health-care information system. An ad-hoc query asking for side-effect of a particular medicine may involve multiple conditions: type of cancer "Lung Cancer", specific medicine "Med2", side-effect "Hair Loss". For example, due to availability of detailed information about patients and their treatments:

1. It can be deduced which treatments are effective on patients or group of patients.

2. Decision regarding diagnosis and treatment will be automated, supported by analysis of previous data.

3. It will also help in mitigating any unintended mistake in diagnosis or treatment.

Pre-eminent solution for implementing HIS is to combine several disparate database

solutions in-line with data storage needs. We are hereby proposing a polyglot-persistent solution for managing the big data of HIS – ***PolyglotHIS***. This polyglot-persistent application uses several database technologies to achieve an amalgamated solution. PolyglotHIS use one relational (PostgreSQL) and two NoSQL data-stores; graph(Neo4j) and document(MongoDB). NoSQL data-stores have been used for storing semi-structured data such as laboratory images and un-structured data such as patient-doctor interaction report, where as relational database is used for storage of structured and financial data.

The coalition of data-stores in PolyglotHIS is not arbitrary, instead it is well planned out with the careful selection of data-stores. As depicted in Figure 5.1 each of the involved data-store has its own specific advantages. Relational data-stores are preferred for data pertaining to financial transactions, since they adhere to ACID properties. Employees' payroll data, patients' billing information and financial component of pharmacy department are handled by relational database – *PostgreSQL*.

In the next section the architecture and underlined components of the proposed PolyglotHIS have been presented in order to have a better insight of the system.

5.2 Architecture and Components

Today healthcare industry is at its wits' end to store variety of data ranging from individual patient/doctor information to patients' medical history including various clinical results. HIS is used to structure this voluminous and disparate stream of data. HIS are multifarious in nature and thus can be best implemented by using multiple data-stores because one database definitely does not satisfy all the storage requirements of such complex applications.

Instead of forcing all types of data to fit in row-column format of relational databases, it is proposed to use multiple data-stores; each storing and representing data closer to its actual representation and usage. Though different departments in hospital deal with diverse type of data and hence require to deploy distinct data-stores. PolyglotHIS free the end-users from interfacing with each data-store individ-

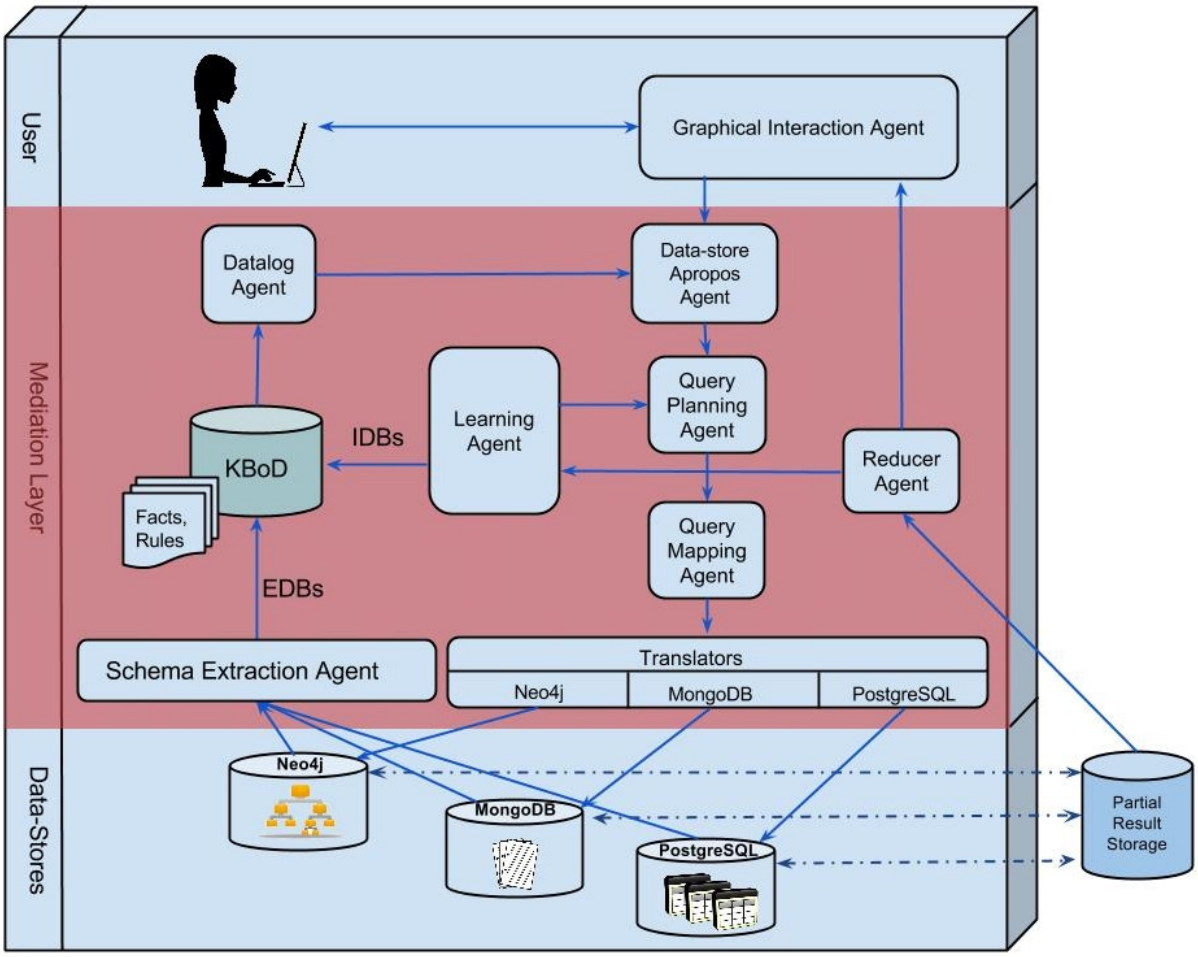

Figure 5.2: Detailed architecture of PolyglotHIS (Proposed System)

ually by facilitating through a uniform query interface facade. Key facet of proposed approach is the use of co-operative multi-agent architecture, together with the use of Datalog for declarative specification of the contents and querying possibilities of underlying data-stores while also assisting the query planning module in trimming trivial data-stores for a query.

The broad goal of PolyglotHIS is to enable access to heterogeneous data-stores, while insulating user from the information pertaining to location and querying mechanisms of underlying data-stores. The simple approach is to construct a global schema that contains snapshot of information present in all data-stores, against which user will pose queries. However, this approach is not free from its associated problems — difficulty in integration of heterogeneous data from multiple and different sources, difficulty in adding new or removing existing data source and requirement of reformulation of the global schema to incorporate any updates in the underlying data-stores.

In place of *data translation*, PolyglotHIS employs *query translation* mechanism, thus eliminating the need of global schema. Query translation has advantages in terms of flexibility, scalability and dynamic query execution. Dynamic query execution takes into account current circumstances such as unavailability of any data-store or addition of a new data-store [149].

The various components underlying the proposed architecture are presented in the next subsections.

5.2.1 Agents

Inter-operation between involved data-stores having different semantics, representation and querying mechanisms has been achieved in PolyglotHIS with the help of various agents. ***Agents*** are software entities which, using artificial intelligence techniques, choose best set of activities to be performed for reaching a particular goal as desired by the user [150]. Agents possess important properties like reacting in a timely fashion, adapting to dynamic and unexpected changes, working autonomously, following proactive approach and having ability to communicate with users and other agents. Intelligence of agents is contributed to their reasoning and learning capabil-

ities. Communication between agents is done using Knowledge Query Manipulation Language (KQML) [151]. Agents have already been widely used in integration systems, especially for enhancing automation [93, 152]. As shown in Figure 5.2, each agent is designated specific task as detailed below.

Query translation approach along with declarative mappings have been used in PolyglotHIS. Query translation takes care of translating query asked using query interface facade into equivalent set of local queries, in specific query language of corresponding data-store. Data is stored in heterogeneous local data-stores. Results obtained from underlying data stores are transmitted to the mediator layer, where they are transformed and combined to be presented to the users or application.

Information stored in mediation layer, also called meta-information, is utilized to divide the global query asked against the common GUI, into multiple sub-queries each asking data from relevant data stores. This meta-information is accessed every time to generate specific local queries. Maintenance of meta-information is very vital for correct working of the PolyglotHIS.

The various types of agents have been presented below:

5.2.1.1 Graphical Interaction Agent

Graphical User Interface (GUI) of PolyglotHIS involves multiple dynamically generated drop-down menus based on querying capabilities of the involved data-stores. It serves as an entry point to the PolyglotHIS with the usage of REST API and also acts as an interface to perform CRUD operations on the data-stores. Aggregated results have also been shown to the user via this agent. Interface has been designed to be simple and easy to use by the users. GUI acts as the front end of the system and let the user query put up a facade over multiple data-stores providing required transparency. Information is presented to the potential users from user's point of view and not in the system's language. Tasks required for navigating through the interface match with real life scenarios for providing a natural flow of actions. Appropriate validations are present when and where required, prompting the user with suitable error messages to correct any error. A minimalist design strategy was fol-

lowed to avoid any unnecessary distraction to the user from the task at hand by avoiding irrelevant and extraneous information.

5.2.1.2 Data-store Apropos Agent

Data-store apropos agent is the primary entry point into the mediation layer of the system and is responsible for determining on-the-fly which data-store(s) is/are required for answering the posed query. This agent performs data-store selection dynamically and identifies the data-stores required to answer the query. It is also responsible for verifying syntactic and semantic soundness of the query. Provision of drop-down driven menus in GUI limits user to ask specific queries but in turn churns out any invalid query posed by mistake by the user of the system. Required attributes asked by the user are passed on to apropos agent to determine *what is to be fetched and from where.*

Data-store apropos agent takes input from *Datalog Agent* and provides processed output to the *Query Planning Agent*. Primary role of apropos agent is to understand the query entered by the user via GUI and check it for syntax and semantics i.e. whether the asked query is feasible to be fetched from the data available in data-stores. The semantic description of data-stores is expressed in form of Datalog facts and rules in KBoD, hence datalog agent retrieves the relevant details pertaining to the query, ascertaining its semantic validity.

5.2.1.3 Datalog Agent

Data-store apropos agent works in conjunction with *Datalog agent*, which asks for the required information from KBoD in the form of Datalog queries. Key obstacle in answering the queries over a collection of data-stores is that the system must understand and utilize the relationship amongst their contents. Meta-data management also assist the system in pruning of data-stores that are not required to answer a specific query. Declarative specification of contents and querying possibilities of underlying data-stores is done using Datalog, which also helps in generation of query

execution plans [153, 154].

Datalog is a nonprocedural query language based on the logic-programming language Prolog [155]. Schema and capabilities of underlying data-stores are exposed to the mediation layer with the help of Datalog facts and rules stored in KBoD which are queried with the help of Datalog agent. Information thus provided by the Datalog agent is utilized for formulating query plans by the Data-store apropos agent.

Datalog extended with skolem functions in the head of rules to handle existentially quantified variables is used in the areas of data exchange and integration, due to resemblance between Datalog rules and GLAV schema mapping mechanisms [154]. Inverse rules algorithm have also been used in integration across multiple source schemas by helping in reformulation of queries against target schema [156]. Existing integration systems that make use of Datalog are Information Manifold [89], Clio [157] and Orchestra [158].

5.2.1.4 KBoD

In order to collaborate amongst the heterogeneous components of a federation, a common model for describing the sharable data must be established. This model must be semantically expressive enough to capture the intended meaning of the conceptual schemas that may reflect several kind of heterogeneities. Further, this model must be simple enough to understand and implement. *KBoD* stands for Knowledge Base of Data and it basically stores semantic description of the contents of each member data-store. It determines where to look for while answering a particular query. Pruning the data-stores not required to answer a specific query is also determined by KBoD. One possible way of constructing KBoD is to store complete semantics of the relationships amongst schemas of all underlying data-stores, using a typical expressive language. In the proposed system, non-procedural query language – Datalog has been used to implement KBoD [153]. Information thus provided by the Datalog agent is utilized for formulating query plans by the Data-store apropos agent. There are several issues in the creation and maintenance of KBoD, such as *How to create it? Is master schema updated every time schema of any underlying data-store*

changes? How information regarding various existing relationships between multiple data-stores is acquired?, which have been explained in detail in Chapter 6.

5.2.1.5 Schema Extraction Agent

Schema extraction agent creates the KBoD in the form of Datalog facts. Extracting schema from schema-less data-stores is one of the most important contributions of this agent. Relational databases are structured databases with mostly fixed schemas that can be easily expressed in any formal model. Expressing schema of NoSQL data-stores is achieved through Datalog. This agent is run first in the whole architecture shown in Figure 5.2 because it is responsible for creating the knowledge base. Then it is run sequentially on all the involved data-stores to build up the Datalog facts. Database administrators then manually intervene to create set of Datalog Rules so that information expressed in form of Datalog facts can be further mined to create new set of Datalog facts.

Extracting schema of structured relational databases is quite trivial. Also, relational databases being matured and used by industry for so long, there are many APIs as well as mechanisms available to express the schema in logic programming. But the same does not prevail for the non-relational databases attributed to their schema-less nature and relative novelty. In order to address this issue, we have designed and implemented a few algorithms to extract and represent the schemas of MongoDB (Document-oriented) and Neo4j (Graph-based) databases.

5.2.1.6 Query Planning Agent

Apropos agent provides the details of data-stores and specific relation/node/document to be queried, to the *Query Planning Agent* which is responsible for constructing best plan for retrieval of data by dividing the query into sub-queries and determining the order of their execution. This component determines how relevant data-stores (identified by apropos module) must be queried in an orderly manner to deduce the desired result. It constructs a plan for data retrieval. Query planner

follows the divide and conquer approach to answer a complex query seeking data from multiple data-stores, dividing the query into a series of sub-queries.

For a given query, multiple query plans are possible. To decide the suitable and efficient plan amongst these possible query plans, *Query Planner* considers various parameters such as overhead of retrieving intermediate results, storing and joining them to produce final data. Besides, the query plan should exploit parallelism if multiple data stores are involved in answering the query, so that sub-queries can be issued concurrently.

5.2.1.7 Learning Agent

Learning agent empowers the query planning agent to find the least expensive query. This plan is derived by execution of previous query plans. Each generated query plan is cached and used for learning by the intelligent information agent based on query performance metrics such as intermediate result storage, total time taken for planning and execution etc. With execution of each query, system is able to learn new rules. These learnings are remembered by the system and looked upon for future queries by embedding this learning into the rules stored in KBoD. It can also optimize the query plan, based on previous query plans and their performances. To make this possible, history of the executed query plans needs to be stored.

Each executed query follows a query plan generated by the query planning agent. This plan is stored in KBoD along with the information received from reducer agent in the form of Datalog rules on top of Datalog facts already stored in the KBoD. Learning of the system is enhanced by execution of each query dealing with underlying data-stores in different manners. Knowledge thus acquired is then fetched by Learning Agent to generate an optimized query plan by the *Query Planning Agent*.

5.2.1.8 Query Mapping Agent

Query plan produced by *Query Planning Agent* is provided to *Query Mapping Agent* to map different parts of the query onto the specific target data-store. PolyglotHIS

consists of more than one data-stores, making the role of mapping agent very crucial. Mostly parallel execution of queries is handled by this component. Order of execution of various sub-queries is also an equally important function of this agent. Ordering of sub-queries plays a very important role in achieving an overall efficiency. Especially, in case of query required to perform join operation, order of joining output of one data-store with another is critically important to plan. Therefore mapping agent plays a very significant role in distribution and execution of the queries.

5.2.1.9 Translators and Partial Result Storage

Translators act as mediators between PolyglotHIS and intrinsic data-stores. Translators receive query written in PolyglotHIS query language, and translate it into required query language of the target data-store. PolyglotHIS is made up of heterogeneous data-stores, each having its own query syntax. More specifically, PostgreSQL uses SQL, while MongoDB uses its own MongoDB query language and Neo4j uses Cypher query language. Existing query builders[1] of PostgreSQL, MongoDB and Cypher are used during the process.

Intermediate data generated by queries is stored in *Partial Result Storage*. For queries using 'join' which require data from multiple data-stores, this type of storage is very essential. Results obtained after query execution are accumulated temporarily in the partial result storage component, that are then passed on to *Reducer Agent* for joining them and providing it back to the user interacting via GUI.

5.2.1.10 Reducer Agent

Reducer agent combines results thus obtained from underlying data-stores available in partial-result storage. Results of various sub-queries are reduced to final result pertaining to one complex query posed by the user at the query interface facade. Integration of results is possible due to availability of JSON format supported by all the involved data-stores. JavaScript Object Notation (JSON) is a human understand-

[1] www.npmjs.com/package/

able text used for data transmission generally stored or transmitted as attribute-value pairs. It has been derived from JavaScript[2]. JSON is highly compact, faster, gets easily loaded into JavaScript and efficient to parse. Results combined by the Reducer Agent are then put up at the GUI for reference of the user. Reducer agent also provide the statistics pertaining to each query to the *Learning Agent* for query optimization purposes.

It is also worth noting that DDL (Data Definition Language) queries affect the schema of the database. Changes in the schema must be updated in KBoD for successful execution of upcoming queries. DDL queries are propagated both ways, towards KBoD for updating master schema as well as updating Datalog rules and underlying data-stores for making the physical changes in the schema. In contrast, DML (Data Manipulation Language) queries update only data present in data-stores. DDL (Data Definition Language) queries affects the schema of the database. Changes in schema must be updated in KBoD for successful execution of upcoming queries.

5.3 Conclusion

In the present scenario a number of options are available to user in terms of data storage and maintenance. One may choose between various relational and non-relational data-stores available. But with the growth of ever complex data and the need to store them optimally, has led to the problem of integrating various data-stores required for storing the data of an application. This can be achieved in many ways. The most feasible way is to store the data of all data-stores in a common format at the background that is understandable by all the data-stores. Here, JSON provides an solution by supporting a format which enhances this interoperability amongst various data-stores.

Polyglot persistent solution takes leverage from highly specialized solutions. Undoubtedly, the overall complexity of the system increases as there is impedance mismatch between various data-stores in terms of the data modeling and the query

[2]JSON (http://www.json.org/fatfree.html)

languages. However, the proposed solution is advantageous because of its ability to store the data according to its usage thereby simplifying the overall programming model.

HIS is a multiplex system dealing with multitude of data and is thus best implemented using multiple database paradigms. Different departments in hospital deal with diverse type of data and hence deploy diverse data-stores. Proposed system empowers integration of these disparate data-stores by providing a uniform query interface facade. Key facet of proposed approach is the use of Datalog for declarative specification of contents and querying possibilities of underlying data-stores that also assist the query planning module in trimming trivial data-stores for a query.

Query translation approach along with declarative mappings have been used in PolyglotHIS. Query translation takes care of translating query posed to query interface facade into equivalent set of local queries, in specific query language of the corresponding data-store. Data is stored in heterogeneous local data-stores. Results obtained from underlying data stores are transmitted to the mediator layer, transformed and combined to be presented to the users or application.

Chapter 6

Design and Maintenance of KBoD

Integration systems that consist of schema-less data-stores like PolyglotHIS face stringent task of extracting schema from schema-less data-stores. Queries asked against PolyglotHIS may require data from more than one data-stores, hence it is of utmost importance to first determine which data-stores are required for each query. For determining this, schema and source capabilities of underlying data sources must be stored in a query-able format.

In PolyglotHIS, schema and capabilities of underlying data-stores are exposed to the mediation layer with the help of Datalog facts and rules stored in KBoD which are queried with the help of Datalog agent. Information stored in KBoD is utilized to divide the global query asked against the common GUI, into multiple sub-queries each asking data from relevant data-store. KBoD is accessed every time to generate specific local queries. Maintenance of KBoD is vital for correct working of an integration system. This chapter provides preliminaries for understanding Datalog and how Datalog facts and rules are used to express schema of the involved data-stores.

6.1 Introduction

Queries posed to PolyglotHIS are reformulated into queries to be asked from multiple individual data-stores. Global query is reformulated into an equivalent set of local queries expressed on the local schemata. The data sources store the real data, while the PolyglotHIS query interface provides an integrated and virtual view of underlying sources. Proposed system retrieves data from multiple sources by performing following tasks — (a) for a given query, determine *which* data attributes are to be retrieved from *which* data source, (b) translate the uniform query into data store specific query, (c) execute queries, (d) unify obtained result and finally (e) present it to the user. Information stored in KBoD also called meta-data, is utilized to divide the global query asked against the common GUI, into multiple sub-queries each asking data from relevant data-stores.

Information stored in data-stores is described logically in the form of views. In literature, multiple approaches for providing integrated view of disparate heterogeneous data sources have been discussed, most popular are LAV (Local-As-View), GAV (Global-As-View) and GLAV which is hybridization of GAV and LAV, also known as TGD (Tuple Generating Dependencies) [159]. In all these approaches, user pose queries to one global mediated schema. Major difference between these approaches is in the way data sources are presented in the global schema. We have considered LAV approach in our architecture, where the decisions about interconnection of data-stores and unification of their data are done during query execution. Mechanism employed for storing description of data-sources must have following properties, in addition to the storage of schema:

1. Ability to find data-sources relevant to a query
2. Express relationship among data-stores *(to enable pruning of irrelevant data-sources)*
3. Addition or removal of data-sources dynamically and easily

Datalog has recently been used in the areas of data exchange and integration, which is due to resemblance between Datalog rules and GLAV schema mapping

mechanisms [154]. KBoD is the brain of PolyglotHIS as it stores the information pertaining to the schemas of the under-lying data-stores in form of Datalog facts (IDBs) and Rules (EDBs). Each datalog rule corresponds to a conjunctive query. Datalog consists of *"if-then"* rules consisting of *facts*, where facts are atomic statements that always hold true. One or more facts together comprise a Datalog *Rule*, which corresponds to *Formula* in propositional logic. A Datalog rule consists of a head and a body and every variable in the head must appear in the body. Name of facts and rules as well as arithmetic symbols are called *predicates*, whose arguments can be variables or constants. Logical variables begin with capital letters, whereas constants can be identifiers or string literals.

Fact-base stored physically on hard disk or memory is referred to as Extensional Database (EDB). Datalog is a rule-based query language which primarily focuses on the rules, making the rule-base inherent to it, thereby named as Intensional Database (IDB). Unlike relational model which can express only facts also known as Extensional Database (EDB), Datalog model can also express Intensional Database (IDB) via Datalog rules. IDBs make use of existing EDBs to define new rules, thereby increases expressing power of Datalog. Existing integration systems that make use of datalog are Information Manifold [89], Clio [157] and Orchestra [158].

Prolog is also a logical language but Datalog is different from Prolog in the following aspects: Datalog requires that the variables appearing in the head of a clause must also appear as positive literals in the body of the clause; it restricts the use of recursion and negations abiding to stratification; it also mandates that a negative literal in the body must have the same positive literal appearing in the body of the clause besides it does not allow complex arguments in predicates. Datalog is a vast language and can be understood well by having prior knowledge of its background and basics of its successors. This chapter discusses First Order Logic, its representation, formal syntax, meaning and interpretation. An introduction to deductive databases is also presented, followed by the introduction to Datalog and its basic constructs. Relationship of established relational algebra with Datalog is also discussed in later sections.

6.1.1 First Order Logic (FOL)

FOL is considered to be the successor of predicate logic with an additional descriptor – *Predicates* in its syntax [160]. Everything is interpreted as either True or False. Formula and predicate are the basic constructs of FOL syntax. For instance "Hector is a frog" is a valid FOL statement where "is a" is predicate and "Hector" and "Frog" are *terms*. Terms can be of three types:

- *Variables*: Example- X, Y, etc. They are further quantified into two categories:-

  Universal quantification: Applicable for all valid substitutions.

  Particular quantification: Also called existential quantification and is applicable for at least one valid substitution.

- *Objects or Concepts*: Constants like Hector, Frog, 1, etc.

- *Functions*: Composed of other terms.

First order language is often written formally in form of a *Quadruple*. The basic signature of FOL is given by:

$$L = (\Gamma, \Omega, \pi, \chi)$$

where L is Language; Γ, Ω, χ are terms of L; and π is predicate of L.

- Γ: Represents all constant symbols.

  It is decidable and non-empty set containing constant symbols.

  Generally represented using character strings as singular entities.

  eg.: a, b, c, 1, 2, Hector, Frog, etc. may represent something or may also contain some values.

- Ω: Represents all function symbols.

  It is disjunctive union ($\cup_{n \in N} \Omega_n$) of the finite sets having n-ary functional symbols.

  eg.: '+', '-', etc. are 2-ary or binary functional symbols.

  eg.: add(x,y), even(x), f(), etc.

- π: Represents all predicate symbols.

  It is disjunctive union ($\cup_{n \in N} \pi_n$) of finite sets having n-ary predicate symbols.

  eg.: P(X, Y), $\{x \mid P(x)\}$, etc.

- χ: Represents enumerable set containing variables.

The language L discussed above is just a syntax pending its interpretation to make some sense out of it. Interpretation needs to find meaning of words, values of variables, working of functions, knowledge of quantifiers and evaluation of other statements in L. So, interpretation captures the semantics of the language L. In order to achieve this interpretation following steps are performed:

- Assign each term to a part of some universe of discourse.
- Assign truth value corresponding to each formula.

Thus interpretation can be written as:

$$\mathrm{I} = (\mathrm{U}, \mathrm{I}_C, \mathrm{I}_F, \mathrm{I}_P)$$

where

- U: universe of discourse,
- I_C: All constant symbols mapped to U,
- I_F: All function symbols mapped to U,
- I_P: All predicates to U.

6.1.2 Deductive Databases

Deducing some kind of knowledge from some predefined facts and rules is the main purpose that deductive databases serve. Deductive database has two major parts:

- ***Extensional database (EDB)***: Knowledge that is external to the system. Refers to the predefined fact base and rules as they are stored externally.
- ***Intensional database (IDB)***: Knowledge that is internal to the database, something derived. Generally refers to the recursive or derived rules.

Deductive databases tend to divide their information/knowledge into two categories [161]:

- ***Data***: Generally called facts or atoms and are represented by constant arguments. Ancestor(Parent, Person) is a fact representing that Parent of a Person is an Ancestor.
- ***Rules***: Generally written as p $\leftarrow p_1, p_2, ..., p_n$. Rules must have a head along with the body. They are declaratively read as $p_1, p_2, ..., p_n$ imply that p is true.

6.2 Datalog

Datalog is a logical query language which is based on deductive database. It consists of "if-then" rules, made up of facts or atoms. A simple datalog program comprises of logical facts (atoms) and rules. Generic representation of an atom in Datalog is as follows:

$$\mathrm{P}(T_1, T_2, ..., T_n)$$

where P is the predicate and $T_1, T_2, ..., T_n$ are terms. Clause or an implication is represented as follows:

$$A_0 \text{ :- } A_1, ..., A_k$$

where A_0 is the *head* and $A_1, ..., A_k$ is the *body*. The clause implies that if all atoms in the body hold true then the head is also true. Also, all the variables that are declared in the head should also appear in the body. This property is also termed as *Rule safety*. Clauses are of two kinds:

- *k=0* → clause is *fact*
- *k>0* → clause is a *rule*

Therefore, it can also be said that *fact* is a clause without body. Datalog efficiently handles recursive queries that makes it more powerful than traditional SQL. A query in Datalog is termed as recursive if there exists a cycle in the predicates' dependency graph. And such queries are evaluated until stagnation or fixed point (when there is no change in consecutive iterations) is reached.

Now consider two facts:

- All numbers that are not even numbers are odd.
- All numbers that are not odd numbers are even.

For this fact program would not be able to make sense out of it, as one depends on the other for its interpretation. This is called *dependency in datalog programs.*

Program Connection Graph (PCG): Another important concept related to datalog programs is PCG. It helps to represent the datalog program in a more understandable manner, and also finds cycles and other dependencies present in that program. While creating a PCG following points must be kept in mind:

- Create a node for each predicate symbol 'p' in the program 'P'.
- Create a directed edge from 'p_1' to 'p_2', if 'p_2' is in the definition of 'p_1'.
- Mark the edge as negative, only if 'p_2' appears as a negative literal, else mark it positive.

Recursive clique is the maximum set of predicate symbols present in program, in a way that there is a path between two predicate symbols in the PCG. A PCG without cycles is hierarchic program whereas a PCG with cycles is a recursive graph.

Stratified: If PCG consists of only positive cycles then the program is termed as stratified. The word stratification means layering. When a PCG has negative edge, then it is generally said that it is on higher layer than all other nodes, thus

introducing layers in the graph and the PCG or the program is called stratified. A datalog program recursive or non-recursive must be stratified, if not stratified then there will be conflict between negative nodes of same layer. The two facts of odd and even stated above are recursive but non stratified and hence not datalog suitable.

Let us consider a simple illustration where three tables are given in the database and some sample queries are represented in relational algebra (RA) format.

Table 6.1: Doctor (D) table with id being #1 and name being #2

id	**name**
1	Dr. Charles Thomson
2	Dr. Gabriel

Name of doctor with id=1?

$\pi_{\#2}\sigma_{\#1=1}(D)$

Table 6.2: Specialization (S) table with id being #1 and specialty being #2

id	**specialty**
1	Dermatologist
2	Surgeon
3	Gynecologist
4	Oncologist

List all specialties of doctor with id=2.

$\pi_{\#5}((\sigma_{\#1=2}D) \bowtie_{(D.\#1=DS.\#1)} DS \bowtie_{([LEFT].\#2=[RIGHT].\#1)} S)$

Table 6.3: Doctor Specialization (DS) table with D_id being #1 and S_id being #2

D_id	**S_id**
1	2
2	1
2	3
1	3
1	4

6.2.1 Datalog to Relational Algebra

For understanding how schema can be represented using datalog, we must first understand how a datalog program can be mapped to relational database with the help of relational algebra. Facts and rules formed in datalog are translated into corresponding relational model entities. We will discuss the whole translation process step by step with the help of examples wherever necessary.

- Translation starts with pre-processing assuming there can not be any constants in the head and are replaced by some variables. These newly created variables are then binded to the old values present in the body. For example,

  p(X,a,b) :- $t_1, ..., t_n$.

  Above written rule is wrong as 'a' and 'b' are constants. After applying, preprocessing we obtain:

  p(X,Y,Z) :- $t_1, ..., t_n$, Y=a, Z=b.

- Variables come at the end while restrictions are listed in the beginning. Evaluation takes place from left to right and thus if constraints are not considered at the beginning they might hamper the filtering. Also the positive literals are listed out first followed by the negative literals. For example,

P :- X=Y, p(X).

Above written rule is not correct as X will hold true for more than one value and p(X) is not considered. This rule is transformed to the rule shown below, which is the correct way of representation.

P :- p(X), X=Y.

- Once the pre-processing is over, translation can be started. Each rule R :- $L_1, ..., L_n$ is transformed to relational algebra using following rules:

  First a relational expression 'E' of atomic components of each literal 'L' in R is transformed.

  $E_i := \sigma_\theta(P_i)$; where θ is called selection criteria and is a conjunction of restraints given as follows:

  For each t_i, condition is added, #j=t$_j$ is a constant symbol and #j=#k if t_j & t_k are same variables.

  For example consider the following rule,

  p(X,2) :- q(X,X,Y,2), r(X,1).

  Pre-processing is required here as there is a constant in the head. Transformed rule is shown below:

  p(X,Z) :- q(X,X,Y,2), r(X,1), Z=2.

  Now X and X are same that means the rule "#j=#k if t_j & t_k are same variables" applies to them. Also "2" in q is a constant which implies the first rule. Let us denote X,X,Y,2 as #1, #2, #3 and #4 respectively and by applying the above said rules, we get 3 expressions respectively for each literal present in P.

  E$_1$:= $\sigma_{(\#1=\#2\ \wedge\ \#4=2)}Q$;

  E$_2$:= $\sigma_{\#2=1}R$;

  E$_3$:= $\sigma_{\#2=2}EQ$; because Z=2 has to be written in its literal form as equals(Z,2).

- Now each expression is evaluated iteratively starting with the first expression as they were processed with each literal from left to right. We need to initialize the temporary expression F as follows:

  $F_1 := E_1$;

  Evaluation of the rest of the expressions and their generation depends on the variables in the literals. If a variable used by the last literal is present or used by the next literal, then it is processed in that order only. Iterations according to constraints:

  $F_i := F_{i-1}$ X E_i;

  Conjunctions of unrelated literals must be computed as their Cartesian product. It is applied only in the cases where L_i does not contain any variables of the last body literals. Eg:

  R := q(X,2), r(Y), EQ(Z,3);

  $E_1 := \sigma_{(\#2=2)} Q$;

  $E_2 :=$ R;

  $E_3 := \sigma_{(\#2=3)} EQ$;

  Now for body expression:

  $F_1 := E_1 := \sigma_{(\#2=2)} Q$;

  $F_2 := F_1$ X $E_2 := (\sigma_{(\#2=2)} Q)$ X R;

  $F_3 := F_2$ X $E_3 := (\sigma_{(\#2=2)} Q)$ X R X $\sigma_{(\#2=3)} EQ$; ;

  Above, we discussed the rules for conjunction of unrelated literals, but in case of related positive literals the result is produced using join employing the related variables as conditions for the join. Thus, in such case body expression can be written as:

  $F_i := F_{i-1} \bowtie_\theta E_i$;

  iff L_i is positive and also shares variables with the last body literals. Eg:

  R := q(3,X), r(Y), X < Y;

R := q(3,X), r(Y), lessthan(X,Y);

$E_1 := \sigma_{(\#1=3)} Q$;

E_2 := R;

E_3 := LT;

For body expression using the above stated rule:

$F_1 := E_1 := \sigma_{(\#1=3)} Q$;

$F_2 := (\sigma_{(\#1=3)} Q)$ X R;

$F_3 := (\sigma_{(\#1=3)} Q$ X R) $\bowtie_{([LEFT].\#2=[RIGHT].\#1 \ \wedge \ [LEFT].\#3=[RIGHT].\#2)} LT$;

The above stated two rules are for positive related and unrelated literals, but, negative related literals can also be there. They are accommodated using set-minus, which removes entities related to negative literals. This is represented as follows:

$F_i := F_{i-1} \backslash (F_{i-1} \ltimes_\theta E_i)$;

iff L_i is negative and has shared variables with last body literals. Eg:

R := q(X), ¬r(X);

E_1 := Q;

E_2 := R;

F_1 := Q;

F_2 := Q $\backslash (Q \ltimes_{(Q.\#1=R.\#1)} R)$;

- After evaluating all the body expressions, the last temporary body expression is algebraically optimized and considered to be final. For instance take the example discussed before the negated literals rule above, after the last body expression the final body expression is written as:

  F := $(\sigma_{(\#1=3)} Q) \bowtie_{(\#2<\#3)}$ R;

- We have evaluated the final iteration result F. Finally the Rule R can now be transformed to expression:

eval(R) := $\pi_{head(R)}$(F);

- There may exist many rules as R discussed above. All the rules have to be processed iteratively through the above mentioned steps. Finally the results of all related rules need to be united which gives the Datalog for the given RA.

  eval(q_i) := $\cup_{R \in def(q_i)}$(R);

In this section we have discussed the basics of First Order Logic, Deductive Databases, Relational Algebra and Datalog. The next section will explain how PolyglotHIS can make use of Datalog. We will discuss creation of KBoD with the help of an example dataset used for MongoDB and Neo4j both.

6.3 PolyglotHIS Knowledge-base of Data

Let us understand how underlying data sources are presented as views and how they are queried, with the help of an example schema. Each involved data-store is accompanied with a set of view(s) defined in terms of global predicates. Along with this, each data-store also have some constraints. The first data-store which supports view v1 provides information about patient, his/her relative's name and contact number. The second data-store gives view v2 which provides information about patient's mobile number and billing amount, provided billing amount is less than $3500. This constraint is important if we ask query about patients whose billing amount is equal to or greater than $3500; then v2 will not be queried. The third data-store contributes view v3 and deliver the patient's symptoms, diagnosis and treatment details. Figure 6.1 represents the explained scenario.

All queries are expressed in terms of a collection of global predicates. For example, if the given query is *"Fetch the name, mobile number and billing amount of all the patients who have undergone the treatment 't'."*, it is formulated using views (representing underlying data-stores) discussed above, instead of using facts directly. For evaluating Datalog queries, bottom-up approach is followed which starts with assessing the facts, followed by binding of each rule with the respective facts. And

finally, executing the query on formed KBoD. The essence of this approach can be best understood with the help of widely used relational algebra. Now we will understand algebraically how the Datalog accomplishes the integration of the disparate views, behind the scene.

```
% Datalog Facts.
patient(P).                 % P is patient
mob(P,M).                   % M is P's mobile number
rel(P,R).                   % R is P's relative
relmob(P,R,RM).             % RM is mobile number of R
symp(P,S).                  % S is symptom of P
diagnosis(S,DI).            % DI is disgnosis of symptom S
treatment(DI,T).            % T is treatment of diagnosis DI
billamt(P,BA).              % BA is billing amount of P

% Datalog Rules.
v1(P,R,RM,M) <= patient(P),rel(P,R),relmob(P,R,RM),mob(P,M).
v2(P,M,BA) <= patient(P), mob(P,M), billamt(P,BA), BA<3500.
v3(P,S,DI,T) <= patient(P), symp(P,S), diagnosis(S,DI),
                treatment(DI,T).

% Datalog Query.
q(P,M,BA) <= v2(P,M,BA), v3(P,S,DI,"t").

% For viewing query results
q(P,M,BA)?
```

Figure 6.1: Datalog facts, rules and queries for an subset of HIS schema.

To transform Datalog facts, rules and query to equivalent relational algebra expression, first all facts are expressed as relational expressions E. Starting with **view v1:**

$$E_1 := \pi_p(pat); \qquad E_2 := \pi_{P,R}(rel); \qquad E_3 := \pi_{P,R,RM}(relmob); \qquad E_4 := \pi_{P,M}(MOB);$$

Then body expression F is evaluated from left to write.

Iteration 1

$$F_{1_1} := E_1;$$
$$F_{1_1} := \pi_P(pat);$$
$$F_{1_1} := pat;$$

Iteration 2

$$F_{1_2} := F_{1_1} \bowtie E_2;$$
$$F_{1_2} := pat \bowtie_{(pat.P\ =\ rel.P)} rel;$$

Iteration 3

$$F_{1_3} := F_{1_2} \bowtie E_3;$$
$$F_{1_3} := pat \bowtie_{(pat.P\ =\ rel.P)} rel \bowtie_{(([left].P\ =\ [right].P)\ \wedge\ ([left].R\ =\ [right].R))} relmob;$$

Iteration 4

$$F_{1_4} := F_{1_3} \bowtie E_4;$$
$$F_{1_4} := pat \bowtie_{(pat.P=rel.P)} rel \bowtie_{(([left].P=[right].P)\wedge([left].R=[right].R))} relmob \bowtie_{([left].P=[right].P)} mob;$$

Similarly for **view 2**

$$F_{2_3} := pat \bowtie_{(pat.P\ =\ mob.P)} mob \bowtie_{(([left].P\ =\ [right].P)\ \wedge\ ([right].BA\ <\ 3500))} billamt;$$

and **view 3**

$$F_{3_4} := pat \bowtie_{(pat.P=symp.P)} symp \bowtie_{([left].S=[right].S)} diagnosis \bowtie_{([left].DI=[right].DI)} treatment;$$

Now, the Datalog query can be expressed as:

$$F_Q := V2 \bowtie_{(V2.P = V3.P)} (\pi_{P,T} (\sigma_{T=\text{“}t\text{”}} V3));$$

Extracting schema from schema-less data-stores – MongoDB and Neo4j is a very challenging process. Once extracted, these schemas must be expressed in a form which can be queried. On the other hand, extracting schemas from Relational database – PostgreSQL being a structured database is very easy. In this section we will focus primarily on how schema from MongoDB and Neo4j is extracted and then how it is expressed in Datalog. Schema-extractors for both MongoDB and Neo4j are written in Python language. As explained in Section 6.2, facts are known as EDBs and rules as IDBs. EDBs are automatically created by the schema-extractors, while IDBs are constructed using extracted EDBs. Useful information about the inter/intra connectivity of data-stores can be inferred from EDBs.

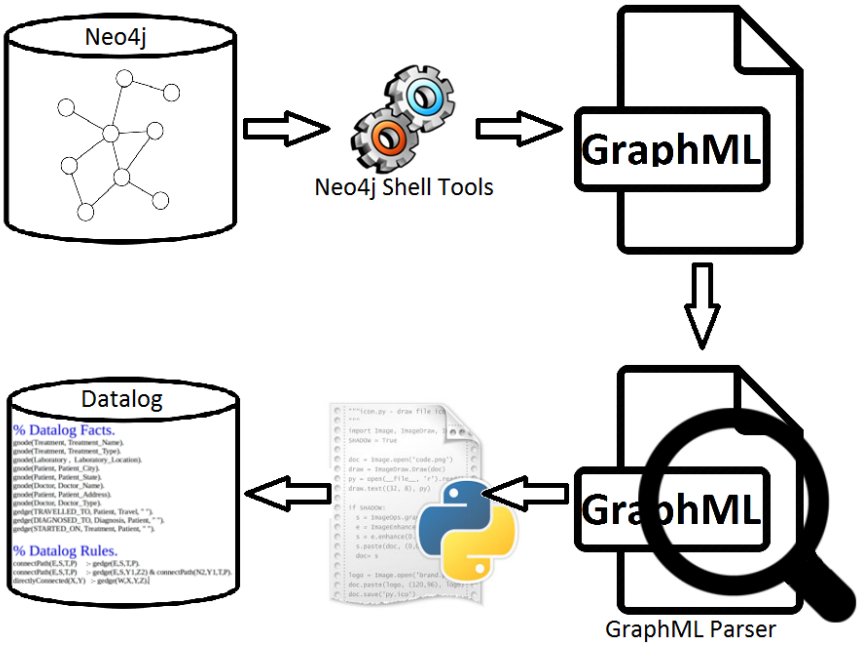

Figure 6.2: Neo4j schema extraction process

The process of schema extraction from Neo4j database is represented graphically in Figure 6.2. Neo4j instance is first exported to GraphML acting as a snapshot of that state of the datastore. Our application takes this snapshot as input and with the help of Python's MiniDom parser, the GraphML file is traversed and the nodes, their attributes, edges and their attributes are stored intermediately in the Python dictionary data structure before storing as facts in Datalog format using pyDatalog.

Unlike Neo4j graph database for which there is no ready made tool available for fetching its schema, fortunately a software named MongoInspector is available for MongoDB which has functionality to extract the schema of underlying MongoDB databases [1]. It helped us in automatic detection of MongoDB collections and fetching documents within each collections. MongoInspector also supports quick browsing of the documents and collections.

As shown in Figure 6.3, there exists individual Datalog facts for each collection in a **DB**, for each document in a **collection** and for each attribute in a **document**. That is, each and every particular details of all databases created in MongoDB is captured in the form of Datalog Facts in KBoD. The fact ***mEmbedDoc*** contains entry for each embedded document in the each database, while fact ***mReferenceDoc*** helps in identifying linked documents within collections of a database. It contains four variables, *first* for the object_id of the document under consideration, *second* for the name of collection it refers to, *third* for the object_id of the referred document in that collecton and *fourth* variable is used to specify if the referred document and collection is presented in a different database, which is generally not the case.

LinkedColls(referringColl, referredColl) ← *mReferenceDoc(id,referredColl,refid), mCollDoc(referringColl,id), mCollDoc(referredColl,refid)* .

Datalog rules written above help in automatic fetching of the list of all collections those refer to each other. This IDB is written with the help of existing EDBs. Information about referencing among collections plays a very vital role when query asked by user is interpreted and unfolded into sub-queries. Similarly, facts and rules

[1] MongoInspector https://github.com/msavin/MongoInspector

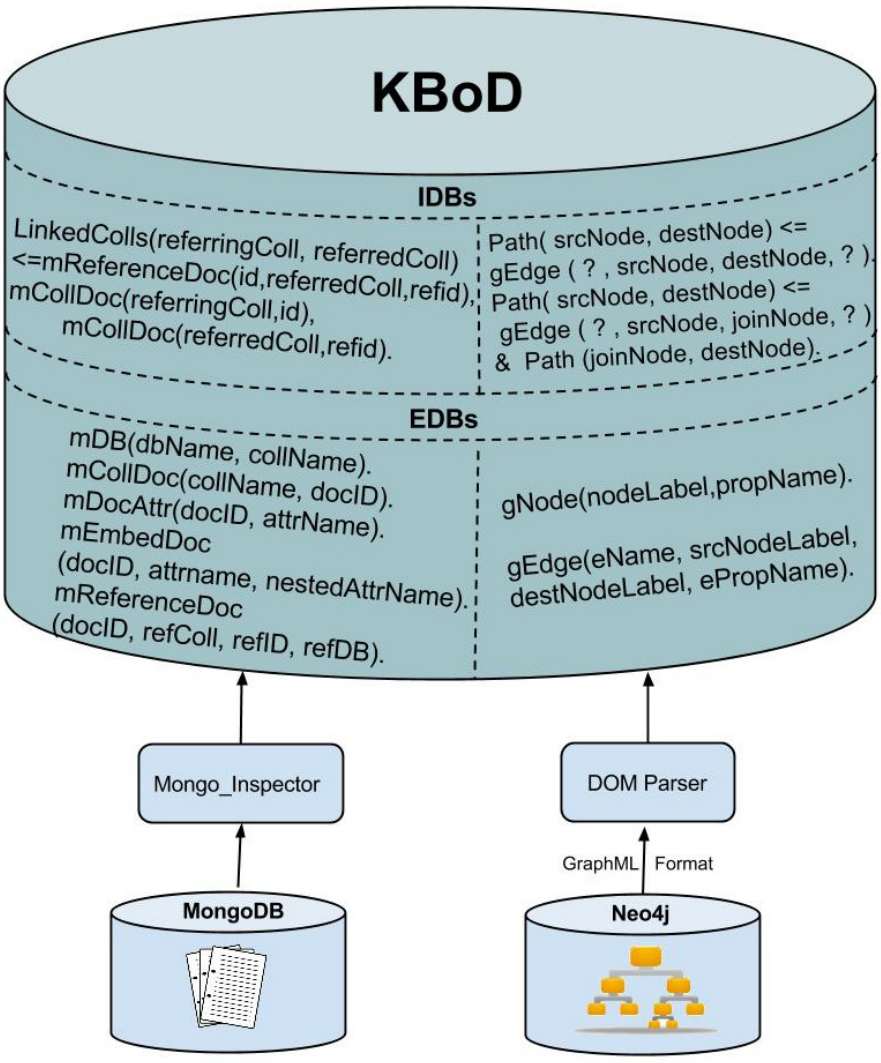

Figure 6.3: Knowledge-base of Data

corresponding to Neo4j graph database are also stored in Datalog. For each pair of graph node and its node properties in the database, one fact **gNode** is created. Likewise, for each edge and each edge-property for that edge, one fact **gEdge** is created in the knowledge base. In addition to storing edge name and property, the source node label and destination node label of the edge is also stored.

Path(srcNode,destNode)←gEdge(?,srcNode,destNode,?).

The above rule make use of fact *eEdge* to form rule *Path*, which in turns provide information about presence of all *direct edges* in the Neo4j graph. *"?"* represents that for all possible values, evaluate the fact and the rules. The above written rule *Path* is further used along with fact *gEdge* to retrieve all *indirect edges* also in the rule written below.

Path(srcNode,destNode)← gEdge(?,srcNode,joinNode,?)& Path(joinNode,destNode).

Visibly, second rule is a recursive rule; Path is calling itself because the first Path finds all direct edges and the second Path uses all possible combinations of these direct edges to ascertain indirect edges.

6.4 Conclusion

This chapter presented the preliminary knowledge on First Order Logic, deductive databases and detailed on Datalog. Transformation of Datalog to relational algebra is also discussed with the help of a suitable example. Schema extraction process for MongoDB and Neo4j data-stores is presented using a subset of database. Schema of Neo4j is first exported in GraphML format using Neo4j shell tools, which is then parsed using DOM parser to store it as Datalog facts. Similarly, schema of MongoDB is first extracted using MongoInspector tool, whose output is then parsed to store details of databases present in MongoDB into Datalog facts. DBAs define Datalog rules such that more insight about the inherent schemas can be gained from the

Datalog facts. Explanation of knowledge-base used by PolyglotHIS is presented in this chapter. Next chapter further explains the working of KBoD with the help of sample data set along with sample query.

Chapter 7

Query and Performance Evaluation of PolyglotHIS

Each of the NoSQL data-store is accompanied with its own query language. Learning and practicing a new query language for each used data-store is a hurdle towards easy adoption of NoSQL data-store in SQL-dominated software industry.

Integration of different NoSQL data-stores is also difficult for the reason that no two data-stores share a common query language. This chapter discusses the need of data integration system for query. A sample query execution have also been showcased to explain the working of the proposed integrated system.

We have also shown example queries of considered NoSQL data-stores in their native query languages. This chapter also elaborates role of KBoD during query formulation process. Performance evaluation of the system is also presented which shows that overhead in terms of latency caused due to presence of multiple layers of PolyglotHIS is very less and becomes negligible as the dataset increases.

7.1 Querying data integration systems

Specialized representation of each data-store is accompanied with specific query language. Each NoSQL data-store comes with its own query language and API bindings with various programming languages like Java, Ruby, Python etc. Learning query language for each NoSQL data-store is an additional overhead, but at the same time provides flexibility in querying underlying data in an expressive way. Absence of a uniform query language for all NoSQL data-stores is one of the biggest hurdles in quick adoption of NoSQL data-stores. Efforts have been made in this direction, but were not very successful[1], due to diverse nature of these data-stores, their querying capabilities and query languages.

Translators are softwares that convert queries into a format that particular local data source understands and also transforms the result set obtained from local data source into appropriate format. Query answering in mediation-based data integration systems is usually performed using following steps:

1. Global query (Q) is asked over global/mediated schema.
2. Master schema is consulted to acquire knowledge about the location of various attributes asked in the query.
3. Q is broken down into sub-queries (local queries) say SQ1, SQ2 and so on, which corresponds to the schemas of local data sources.
4. Subqueries are further optimized and a query plan is constructed for each local query.
5. Query plans made in Step 4 are then executed on local data sources.
6. Results obtained from step 5 are combined according to global query (Q) plan.

Query processing in Local-As-View (LAV) approach is difficult as compared to Global-As-View approach because views in LAV provide very less knowledge about data that conforms to global schema [162].

[1]UnQL (http://unql.sqlite.org/)

7.1.1 Query Languages

Developer community is well versed with the usage of Structured Query Language (SQL) for storing, retrieving and manipulating data in relational databases. As of now, there is no standard query language available for querying non-relational databases. Querying these new generation of databases is data-model specific. Each database comes with its own query language e.g. CQL (Cassandra Query Language) for Cassandra, MongoDB Query Language for MongoDB, Cypher Query Language for Neo4j etc. Efforts are in progress to design query languages which can be used by various databases. But designing a query language that can span multiple data-models is difficult because as explained in previous sections, each classes of NoSQL database is designed for some specific purpose. Hence, efforts are being made to design a common query language for NoSQL databases have been localized to its classes. For example, SPARQL is a declarative query specification designed for most of the available graph databases [60]. Similarly, UnQL (Unstructured Query Language) was introduced in 2011 by Couchbase and SQLite team with the aim to create a standard for NoSQL database queries. It has SQL-like syntax for manipulating document databases and can be used across various document-oriented databases including CouchDB and MongoDB [116].

Most of the NoSQL databases allow RESTful interfaces to the data, many others offer query APIs. Few query tools/editors have also been developed for respective NoSQL databases. MongoDB has its own query language. The find() method selects documents from a collection that meets the ⟨query⟩ argument. ⟨Projection⟩ argument can also be passed to select the fields to be included in result set. The find() method returns a cursor to the results which can be assigned to a variable. Query syntax looks like *db.collection.find*(⟨*query*⟩,⟨*projection*⟩). The find() method is analogous to the SELECT statement, while the ⟨query⟩ argument corresponds to the WHERE statement, and the ⟨projection⟩ argument corresponds to the list of fields to select from the result set [163]. Another method findOne() is just like find() except that it selects only one document from a collection.

Cypher is a declarative graph query language which is used to query Neo4j graph

database. It allows expressive and efficient querying and updating of the graph store without writing traversals through the graph structure in code. Cypher is designed to be a humane query language, suitable for both developers and professionals who want to make ad-hoc queries on the database. Cypher is inspired by a number of different approaches and builds upon established practices for expressive querying. Most of the keywords like WHERE and ORDER BY are inspired by SQL [164]. Pattern matching borrows expression approaches from SPARQL. Being a declarative language, Cypher focuses on the clarity of expressing what to retrieve from a graph, not how to do it, in contrast to imperative languages like Java, scripting languages and JRuby Neo4j bindings.

Like SQL, Cypher is a declarative language. The syntax is easy to comprehend due to usage of clauses similar to SQL like start, match, where, return etc. Usage of wild characters like *,? enhances pattern matching. The command *order by* is used similar to SQL. Almost all aggregate functions like count, min, max, avg, distinct etc. are also available in this language. Presence of predicates like all, any, none, single, increases querying capabilities of the language. SQL literate developers can easily use Cypher query language [165].

Table 7.1 shows query syntax of three databases, PostgreSQL for relational database, MongoDB query language for MongoDB and Cypher query language for Neo4j. Three queries with different complexities have been considered here. Queries are chosen in such a way that maximum syntax of query languages are covered.

Table 7.1: List the patients of each doctor.

PostgreSQL	*select p.patid, d.patid from patient p, doctor d where p.patid = d.patid;*
MongoDB	*db.doctor.find({},{_id:1,patid:1})*
Neo4j	*start n=node(2,3,4) match n⟨-[:IS_A]-f⟨-[:TREATED_BY]-g where n.name='patient' return f.uid as userid, g.pid as posted by*

Q1 : List the patients of each doctor.

MongoDB: 1 is put against both _id (patient-id) and did (doctor-id) which implies that for each patid of Patients collection, all *dids* should be returned.
Neo4j: 2,3,4 are node-ids similar to row-ids of relational databases. Multiple nodes are selected here (2,3 and 4) as starting point. A starting point is a relationship or a node where a pattern is anchored. Here, g is the post TREATED_BY f which IS_A node with name 'doctor'.

Table 7.2: Find the medicines which have been prescribed in the treatment of each patient.

PostgreSQL	*select p.patid, m.medid from patient p, treatment t, medicine m where p.patid = t.patid and m.patid = t.pid;*
MongoDB	*db.treatments.find({},_id:0,patid:1,medid:1)*
Neo4j	*start n=node(2,3,4) match n⟨-[:IS_A]-f⟨-[:TREATED_BY]-g-[:HAS_PRESCRIBED]-⟩h where n.name='patient' return f.patid as patientid, h.medid as medicineid*

Q2 : Find the medicines which have been prescribed in the treatment of each patient.

MongoDB: 0 is placed against _id which implies that treatmentids is not to be projected in the answer.
Neo4j: This query is same as above, just one more relationship HAS_PRESCRIBED has been added.

Q3 : Find the patient names which are in the same ward where patient "pat1" is admitted.
MongoDB: Document with patname 'pat1' is extracted from Patients collection and stored in variable p. Then ward corresponding to that patient is found by

Table 7.3: Find the patient names which are in the same ward where patient "pat1" is admitted.

PostgreSQL	*select p.name from admit a, ward w, patient p where w.pid = a.patid and a.patid = p.patid and p.patname = 'pat1';*
MongoDB	*var p = db.Patients.findOne(patname:"pat1"); var ward = db.Ward.findOne("admit.patid":p._id); db.Patients.find(_id:$in:Patients.patid,_id:0,name:1*
Neo4j	*start n=node(2,3,4) match n⟨-[:IS_A]-f⟨-[:ADMITTED _BY]-g-[:WARD_IN]-⟩h-[:HAS_ PATIENT]-⟩k where n.name = 'patient' and f.patname = 'pat1' return k.name*

matching p._id field with patid field of subdocument admit (admit.patid) in Ward Collection and returned cursor is stored in the variable ward. Then _id is matched in Patients collection. To exclude _id field, _id:0 is written in projection field and the fields which need to be included are written as fieldName: 1.

Tables 7.1, 7.2 and 7.3 provides a quick introduction to syntax of three data-stores used in PolyglotHIS with the help of sample queries. A brief explanation of syntax of MongoDB and Neo4j is also given along. We have not briefed on the query syntax of PostgreSQL, since SQL is the most popular database query language and is being used by everyone everywhere.

7.2 PolyglotHIS Demonstration

Seamless inter-operation between multiple data-stores storing different information pertaining to patients and other important functioning of healthcare information system is difficult to achieve due to usage of different data models and query languages. Different database paradigms are tuned to meet storage needs of various use cases. For example, if requirement is quick storage and retrieval: document data-store is preferred, where as for performing complex analysis on data: column-

oriented databases are more suitable. Graph databases provide features for tracking and managing complex relationships. One popular database of each database model was chosen for the implementation, PostgreSQL for relational database category, MongoDB for document-oriented NoSQL data-store and Neo4j for graph data-store.

To get the best out of all available databases, data is partitioned across three types of databases. Data related to person is stored in ***PostgreSQL*** because the application must maintain confidentiality and security of patient's personal information. PostgreSQL easily integrates with the security framework provided by the technologies used to build any application, and so it provides the ideal repository for holding the details of employees and their passwords. Data pertaining to hospital payroll and patient billing is also stored in PostgreSQL to meet OLTP requirements of the system. Financial data is stored in relational databases because of their in-built support for ACID properties. Data related to active billing is stored in PostgreSQL, where as historical record is stored in MongoDB document database as a series of billing history documents. Each billing document has fully de-normalized structure to make complete history to be retrieved easily and quickly. Time-stamp of billing transaction distinguishes different bill records.

Laboratory reports and images are stored in ***MongoDB***. Information pertaining to every medical event i.e interaction between doctor and patient contains flexible information so it is stored in document database. Manuals of instruments, photo of patients and doctors, history of symptoms, bill details etc. are also stored in MongoDB. Information about medicinal history which varies from patient to patient and may contain different attributes for different patients is schema-less and is thus best suited to be stored in MongoDB again. While, blood relationship between various patients is stored in Neo4j graph data-store, which helps a doctor to trace any hereditary diseases. Inter-linking between symptoms has also been stored in ***Neo4j*** graph database to help a doctor to visualize the links between symptoms and to diagnose quickly about the possible disease. Salt configuration of various medicines is also stored in the form of graph in Neo4j.

7.2.1 MongoDB Data-set

MongoDB stores data in the form of documents which are grouped together in collections. MongoDB database is made up of various collections. Compared to relational databases, collections correspond to tables and documents to records. Unlike relational databases where every record in a table has the same number of fields, documents in a collection can have completely different fields. Documents are addressed in the database via a unique key called *object_id* that represents the document.

NoSQL data-stores do not support *joins*. MongoDB supports two ways of linking documents – *Embedding* and *Referencing*. Embedding is achieved through nested documents whereas in referencing, details of referenced document – $db, $ref(referenced collection name) and $id(of referenced document) is provided in the referring document. A subset of data stored in MongoDB is shown in Figure 7.1 for patients and doctors collection respectively. PatMedication of Patients collection is an embedded document, whereas DocPatientIds of Doctors collection is an referenced document.

Unlike relational databases, where joins can be easily identified by listing primary keys and foreign keys of the tables; extracting relation between documents of a collection is not a simple task in MongoDB. Identifying referencing among documents is done by checking the value of attributes, if the value is of type object_id and that object_id exists as key in some other document, then a linking between documents is established. This linking (referencing) among documents is expressed in the form of Datalog rules which are useful during query re-writing process. Information stored in MongoDB documents and matching Datalog facts have one-to-one correspondence.

Description of each database, constituent collections and enclosed documents and their attributes is stored in form of Datalog facts, while relation among collections is found out with the help of Datalog rules. MongoDB schema extractor with the help of *mongo_inspector* extracts the basic schema of MongoDB and pt it in the form of Datalog facts in the KBoD. These facts can be queried in various ways to extract useful information about the schema of MongoDB. Datalog agent execute suitable Datalog queries when asked by the data-store apropos module about the necessary schema elements to formulate a query plan. Datalog rules provides additional power

```
> db.Patients.find().pretty()
{
"_id":"OCLS1326",
"Name":"P1",
"Gender":"M",
"DOB":"18/07/1976",
"BloodType":"A-",
"Address":{
        "HNo":"65",
        "StreetNo":"3",
        "Locality":"Officers
            Colony",
        "City":"Patiala",
        "State":"Punjab",
        "PinCode":"147001"
        },
"ContactDetails":{
        "MobNo":"9814725632",
        "ResNo":"0175-2353587",
        "EmailId":"p1@gmail.com"
        },
"DrugAllergies":[
        "Aspirin","Penicillin"
        ],
"Medication":[{
        "Name":"Med1",
        "StartDt":"15/02/2014",
        "EndDt":"17/04/2014",
        "Freq":"2 Times a Day",
        "Dose":"10mg",
        "SideEffect":"Nausea",
        "PrescribedBy":"SCL054"
        },
        {
        "Name":"Med2",
        "StartDt":"22/04/2014",
        "Freq":"2 Times a Day",
        "Dose":"10mg",
        "PrescribedBy":"SCL130"
        }]
}
```

```
> db.Doctors.find().pretty()
{
"_id":"SCL130",
"Name":"D1",
"Gender":"M",
"DOB":"15/08/1971",
"Speciality":{
        "Type":"Surgeon",
        "Area":"Thoracic"
        },
"Education":"M.D.(Thoracic)",
"Certifications":[
        "Surgery",
        "Thoracic and Cardiac
            Surgery"
        ],
"PatientIds":[
        "OCLS1326",
        "DCMN9101",
        "LMNH459"
        ],
"BloodType":"O+",
"Location":{
        "Hospital":"Fortis",
        "City":"Mohali",
        "State":"Punjab",
        "PinCode":"160162"
        },
"ContactDetails":{
        "MobNo":"9865329854",
        "OfcNo":"0172-5006942",
        "EmailId":"d1.
            fortis@gmail.com"
        }
}
```

Figure 7.1: Fragment of Patients and Doctors Collections

```
% --- MongoDB Datalog Facts ---
mDB(HIS,Patients).                          % mDB(dbName, collName).
mCollDoc(Patients,"OCLS1326").              % mCollDoc(collName, docID).
mDocAttr("OCLS1326",PatName).               % mDocAttr(docID, attrName).
mDocAttr("M",PatGender).                    % mDocAttr(docID, attrName).
mDocAttr("A-",PatBloodType).                % mDocAttr(docID, attrName).
mDocAttr("Aspirin",PatDrugAllergies).       % mDocAttr(docID, attrName).
mDocAttr("Penicillin",PatDrugAllergies).% mDocAttr(docID, attrName).
.
.
% mEmbedDoc(docID,attrName, nestedAttrName).
mEmbedDoc("OCLS1326",PatMedication,_id).
mEmbedDoc("OCLS1326",PatMedication,Name).
.
.
% mReferenceDoc(docID, refColl, refID, refDB).
mReferenceDoc("SCL130",Patients,"OCLS1326",HIS).
mReferenceDoc("SCL130",Patients,"DCMN9101",HIS).
.
.
% --- MongoDB Datalog Rules ---
LinkedColls(referringColl,referredColl)<=
                    mReferenceDoc(id,referredColl,refid),
                    mCollDoc(referringColl,id),
                    mCollDoc(referredColl,refid).
% --- For viewing query results ---
?LinkedColls(referringColl, referredColl)
% --- Result ---
referringColl    |     referredColl
-----------------|------------------
Doctors          |         Patients
Patients         |         Treatments
.
.
```

Figure 7.2: MongoDB Datalog Facts and Rules

to the KBoD by expressing hidden knowledge about schema obtained from basic Datalog facts.

Figure 7.2 demonstrates how Datalog facts and rules are stored on KBoD for data-set shown in Figure 7.1. HIS is the database name which contains "Doctors" collection. One document in collection "Doctors" have object_id "SCL130". Further, entry of *mDocAttr* fact exists in knowledge base for each attribute in the Document. Information about embedded documents is stored with the help of *mEmbedDoc* facts. Similar to mDocAttr, mEmbedDoc will store information about each attribute of the embedded document. Extracting information about *referencing* between documents and storing them in knowledge-base is very essential for query decomposition. For each referenced document, one *mReferenceDoc* fact is created in KBoD and it contains information about calling document's object_id, called document's collection name, called document's object_id and called document's database name.

7.2.2 Neo4j Data-set

Graph datastores efficiently cut down the extensive costs incurred in the traditional database transaction system due to the joins [53]. Nodes in graph datastore have direct pointers to all the connected nodes and hence avoid the index lookups as well. As they do not possess a rigid schema or demand a compliance to one, they are highly suitable for representation of dynamic and evolving data. Cycles in graph datastores simplify the complex implementation of self-join or self-referencing entities in relational model. Very largely connected data can make greater sense and produce better understanding in graph visualization than that in tabular representation.

Neo4j models the database as a network structure containing nodes and edges relating nodes to represent relationship amongst them. Nodes may also contain properties that describe the real data contained within each object. Similarly, edges may also have their own properties. Relationship connects two nodes and may be directed, direction adds meaning to the relationship. Comparing with Entity-Relationship Model, a node corresponds to an entity, property of a node to an attribute and relationship between entities to relationship between nodes. Figure 7.3 shows a subset

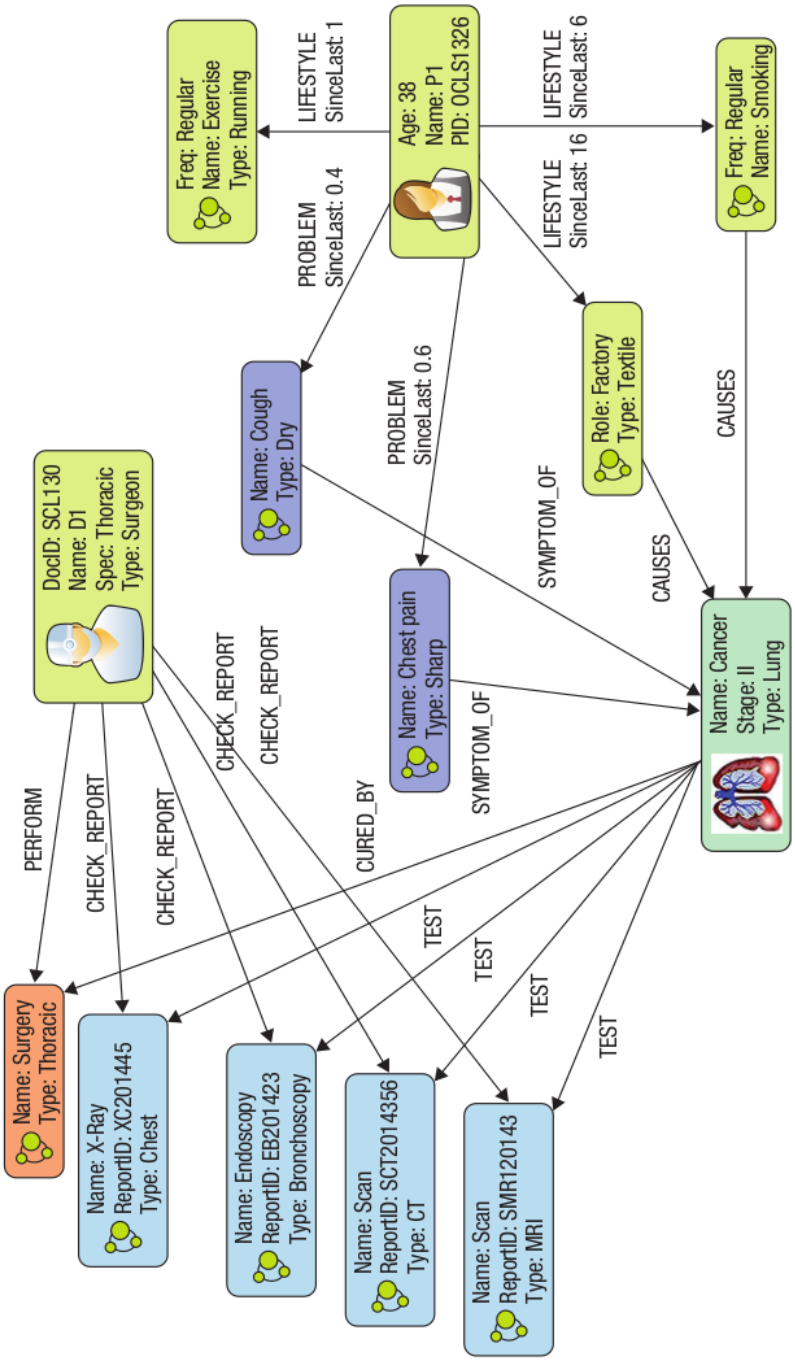

Figure 7.3: Subset of Neo4j graph database shown using Neoclipse editor

of functionalities of healthcare database which are stored in graph database Neo4j. The graph is drawn using Neoclipse editor.

Neo4j v1.0, the very first edition of the database was released officially in February, 2010. In 2012, an intermediate release v1.6 was made to improvise an already reliable product and make it even more reliable. Many intermediate versions were released and then v1.9 was released on May 13th, 2013, introducing extensive improvements in stability. Other than performance improvements, Neo4j v1.9 also featured auto-clustering. In December 2013, Neo4j 2.0 which was in the development process for long was finally released. The user interface (UI) was completely rewritten. Cypher query language was also changed in Neo4j 2.0. It was made far more declarative and concise in Neo4j 2.0. The new feature such as extended support for labels and indexes have been added Neo4j which means that now nodes can be labeled and the properties can be indexed.

The database shown depicts interaction between patient and doctor and how that data is being stored in a database. The nodes have their own properties which are stored in form of key-value pairs. Each node can be given a label to call with later during querying. Same is true for relationships. The edges define relationships and relationship types. They can also have properties which are also stored as key-value pairs. As depicted in Figure 7.3, the edges are directed and hence signify an additional information about the relationship between the two nodes it joins. Edges may be bi-directional as well, but are declared using two uni-directional edges to and fro between edges. So directional adds another dimension to the features of edges, such as Surgery is "performed by" a Doctor is an edge directed from Surgery to Doctor.

Information about each node and edge in Neo4j graph is expressed in form of Datalog facts as shown in Figure 7.4. Path between a pair of source-destination node is calculated using Datalog rules made of Datalog facts. List of all direct and in-direct paths between all pair of nodes considered in the example schema shown in Figure 7.3 is shown as result. *gNode(Patient,Name).* is a fact about Neo4j schema stating that there is a node in Neo4j with label *Patient* and property *Name*. Similarly, another fact *gEdge("PROBLEM",Patient,Symptom,SinceLast).* is an edge in Neo4j

```
% --- Neo4j Datalog Facts ---
% gNode(nodeLabel,propName).
gNode(Patient,Name)
.
.
% gEdge(eName,srcNodeLabel,destNodeLabel,ePropName).
gEdge("PROBLEM",Cure,Symptom,SinceLast)
.
.
% --- Neo4j Datalog Rules ---
% Direct edge
Path(srcNode,destNode) <= gEdge(?,srcNode,destNode,?).
% In-direct edge
Path(srcNode,destNode) <= gEdge(?,srcNode,joinNode,?) &
                          Path(joinNode,destNode).

% --- For viewing query results ---
?Path(srcNode,destNode)
% --- Result ---
srcNode    | destNode
-----------|-----------
Patient    | Lifestyle
Patient    | Symptom
Patient    | Work
Doctor     | Test
Lifestyle  | Disease
Doctor     | Cure
Symptom    | Disease
Cure       | Doctor
Work       | Disease
Disease    | Cure
.
.
```

Figure 7.4: Neo4j Datalog Facts and Rules

with edgename *PROBLEM*, source and destination nodes are *Patient* and *Symptom* respectively and *SinceLast* is the edge property.

7.2.3 Sample Query Execution

Let us understand the usage of KBoD by the Datalog agent with the help of following example query.

Q: *Get the names of all the patients who are have pending dues and have been prescribed "Med2" after being operated for "Lung Cancer" and had side effects like "Hair Loss".*

For this query, data is required from all the three underlying data-stores – relational, document and graph data-store.

1. **From Neo4j:** all patient-IDs who had *lung cancer* and was *cured by* performing *throacic surgery*, and the doctor-ID who *performed* the surgery is retrieved.
2. **From MongoDB:** for each pair of patient-ID and doctor-ID obtained from above query, return those patient-IDs whose *medication* is *Med2* and medication's *side effect* is *hair loss*.
3. **From PostgreSQL:** details of patients out of all patient-IDs returned from MongoDB having pending dues is retrieved.

The steps of execution are visualized in Figure 7.5.

Graphical User Interface (GUI) of PolyglotHIS involves multiple dynamically generated drop-down menus based on querying capabilities of involved data-stores. User is asked to select the attributes he/she want to know and also provide few bare minimum input values for querying the system. For example for the above query, user selects the patient's name as required attribute and provide details such as condition of "Dues Pending", medicine prescription "Med2", disease "Lung Cancer" and side-effects "Hair Loss" from respective drop-down menus.

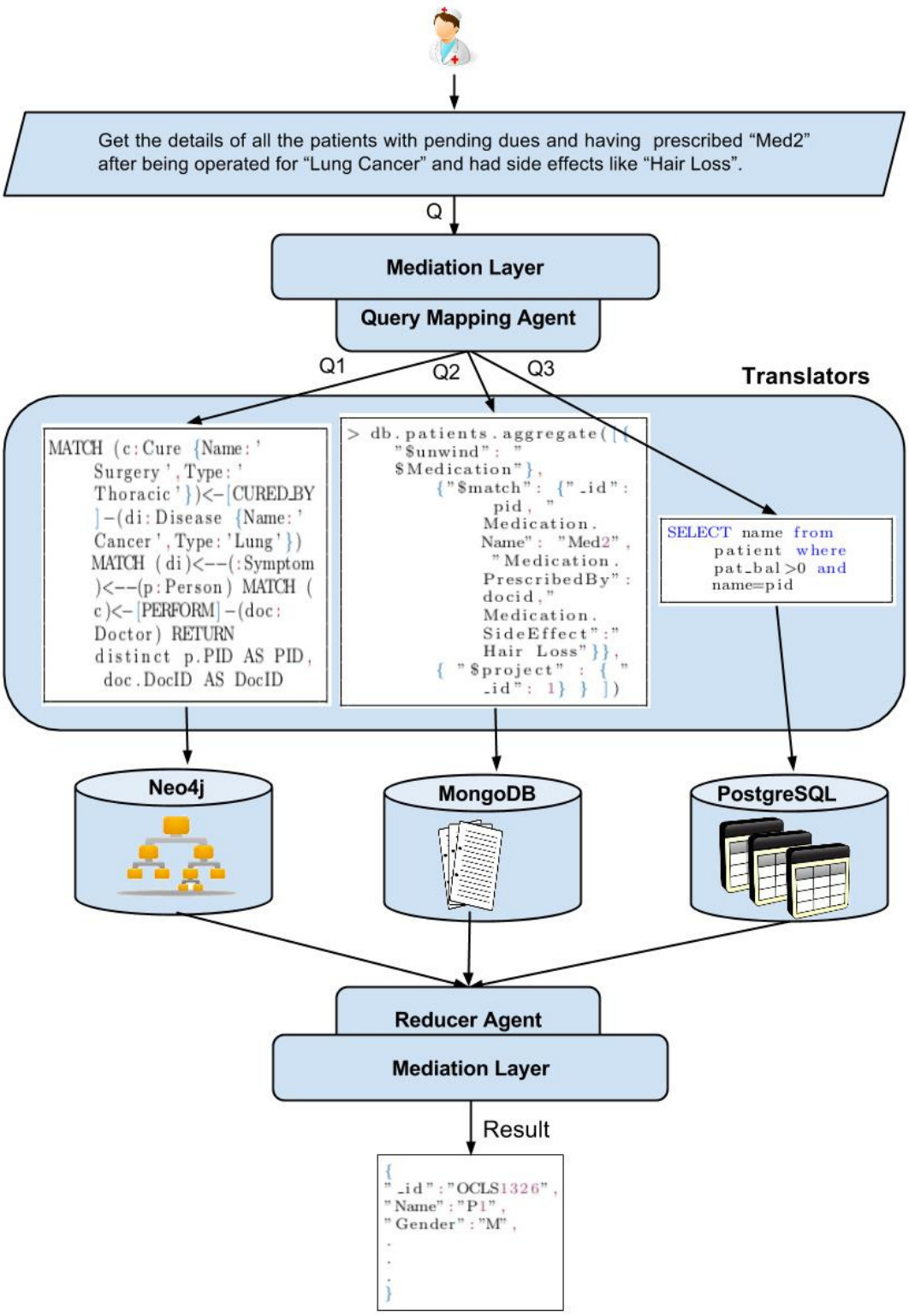

Figure 7.5: PolyglotHIS Query Processing Example

At this point, belongingness of particular attributes to the respective data-stores can be determined at the programming end. Details of disease are available in Neo4j, medication details and side-effects information is available in MongoDB and information concerning dues is stored in PostgreSQL. Data-store apropos module asks the Datalog agent to determine the exact table/document/nodes to be queried as per the user's query requirements. The process of retrieving schema information from Datalog to translation of queries is shown in Figure 7.6.

Final sub-queries executed on under-lying data-stores is in their native querying syntax; translated using query translators which in turn uses existing query builders. Variables obtained from Datalog facts and rules are used in query formulation. For example in Neo4j, attributes made available by the user in query, are searched in Neo4j Datalog facts to retrieve the node and edge labels to be queried. Similarly in MongoDB and PostgreSQL data-stores. In Neo4j, connectivity between asked nodes and edges is also obtained, since Neo4j query is traversal based unlike MongoDB and PostgreSQL queries. MongoDB data-stores stores data in following hierarchy: DB - Collections - Documents - Attributes. User provides only attribute details in the query, remaining details of hierarchy is back-tracked using Datalog fact base.

Query considered for explanation purpose here, is executed sequentially along the data-stores. If sub-queries are not dependent and can be executed in parallel, final result will be obtained by joining the result in JSON format. This is attributed to the fact that all data-stores considered in PolyglotHIS supports JSON format. Order of execution of sub-queries is crucial for retrieving correct results as well as for generation of optimized query execution plan, it is taken care of by planning agent.

```
% NEO4J

% Fetching Node Labels
NodeLabel(D)<-gNode(D, Disease).
NodeLabel(C)<-gNode(C, Cure).
NodeLabel(P)<-gNode(P,PID).
NodeLabel(DOC)<-gNode(DOC, DocID).
% Fetching their connectivity
?Path(D,C). % Connected
     D       |       C
-------------|-------------
Disease      | Cure
% Fetch edge label and property
Edge(CB,CBP)<-gEdge(CB,D,C,CBP).
?Path(D,P). % Not connected
?Path(D,DOC). % Not connected
.
.
?Path(C,DOC). % Connected
     C       |       DOC
-------------|-------------
    Cure     |    Doctor
% Fetch edge label and property
Edge(PB,PBP)<-gEdge(PB,C,DOC,PBP).
```

Final Cypher Query:

```
MATCH(c:C{Name:"Surgery",Type
    :"Thoracic"})<-[CB]-(di:D{
    Name:"Cancer",Type:"Lung"}
    )
MATCH(di)<--(:Symptom)<--(p:P
    )
MATCH(c)<-[PB]-(doc:DOC)
RETURN p.PID AS PID,doc.DocID
    AS DocID
```

```
% MONGODB

CollName(PC)<-mCollDoc(PC,PID).
% PC willl contain name of
    collection with _id=PID (
    obtained from Neo4j)
CollName(DC)<-mCollDoc(DC,DocID).
DBName(PDB)<-mDB(PDB,PC).
DBName(DDB)<-mDB(DDB,DC).
mDBCollDoc(DB,C,DOC)<- mDB(DB,C),
mCollDoc(C,DOC),mDocAttr(DOC,"
    MedName").
```

Final MongoDB Query:

```
> PDB.PC.aggregate(
[{"$unwind": "$Medication"},
 {"$match":
  {"_id": PID,
  "Medication.Name": "Med2",
  "Medication.PrescribedBy":
  DocID,
  "Medication.SideEffect":
  "Hair Loss"}
 },
 {"$project" : { "_id": 1}}
])
```

```
% POSTGRESQL

RTable(X)<-rtable(X,"pat_bal").
RTable(Y)<-rtable(Y,"pat_name").
RTable(Z)<-rtable(Z,"pat_id").
% X,Y and Z all variables contain
    same table name, hence no join.
```

Final PostgreSQL Query:

```
SELECT Z.pat_name from X where X.
    pat_bal>0 and pat_id=PID;
```

Figure 7.6: Sample Query Execution using Datalog fact base

7.3 Experimental Analysis

This section evaluates the implemented multi-paradigm framework in terms of query execution time. The evaluation showcases the performance of the system in terms of total time spent in processing queries over large data-sets. The graphs shown below highlights the overall trend and the weak as well as strong points of each of the three data-stores in comparison to the implemented framework as the data size increases. We have used the total time spent for query processing as our performance metric for individual data-stores. For assessing PolyglotHIS, we have considered the total time spent at the mediation layer for each query.

Python was used for implementing the PolyglotHIS as well as for querying the data-stores individually. Python is a high-level programming language developed by Guido Van Rossum in late 1980s, with motive to add some features such as exception handling, etc. to then prevalent programming languages. Its ideology is based on keeping it really simple, highly understandable, readable, etc. Most of the programming paradigms such as object-oriented programming, imperative and functional programming, automatic garbage collection, etc. are supported by Python [166].

Python can work across various platforms, also third party tools make it really easy to package modules developed in Python to export to standalone executables deployable on other operating systems. Python comes bundled with a default python development environment **IDLE** (Integrated DeveLopment Environment) which is cross-platform clutter-free application for easy development. As indenting is of essence to Python IDLE achieves this with greater simplicity and with its auto-completion feature it comes in handy to the developer. Moreover, it support multiple instances, syntax highlighting, integrated debugger, etc which makes it a strong alternative to traditional terminal or notepad style editing.

pyDatalog was used for implementing Datalog functionality using Python. pyDatalog can be used to perform recursive queries efficiently, to simulate intelligent behavior with the help of rules and learning, to query complex related data as in data integration, etc. Whenever a query is encountered a resolution engine searches

for related clauses and then determines accordingly the best path to follow or which clauses to consider to reach to the goal.

Python and Datalog together give an upper edge as the programmer can choose to use any of the python aggregation functions such as sum, avg, etc along with the facts and rules of Datalog. Python in itself is highly efficient, Datalog is capable of performing some calculations or implementing some logics in lines even fewer than Python, thus together they lessen the execution time and increase the performance manifolds. For instance pyDatalog is capable of solving 8-queen problem in just 0.3 seconds.

Following are the configuration details of the machine used for implementation:

- *Operating System*: Ubuntu 14.04 LTS
- *OS type*: 64-bit
- *RAM*: 8GB
- *HDD*: 700GB
- *Processor*: Intel Core i7-4700MQ
- *CPU clock rate*: 2.40GHz
- *CPU cores*: 8
- *Graphics Processor*: NVIDIA GeForce GT 755M/PCIe/SSE2 (2GB)

Experiments were performed to compare the performance of PolyglotHIS with its intrinsic counterparts. The goal was to determine the overhead introduced by multiple layers of the integration system. Experiments were executed on a single machine having MongoDB 2.4.11, PostgreSQL 9.1.14, Neo4j 2.1.3 and Python 2.7.3. Performance of PolyglotHIS will not vary heavily upon running in more complex configuration since it runs locally and in-memory. Results shown here should not be used for comparing the three native data-stores amongst themselves, since we have

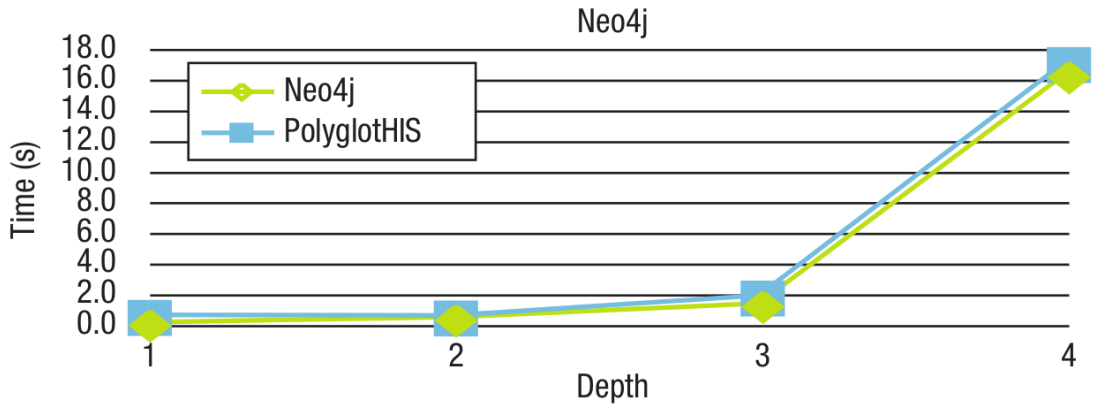

Figure 7.7: Performance comparison of PolyglotHIS with Neo4j for retrieval operation

used different queries for each of them. Focus is on guaging the amount of overhead incurred due to the presence of multiple layers of PolyglotHIS.

We have compared the performance of an hospital information system (HIS) using PolyglotHIS and same system using only the underlying data-stores. We have implemented four editions of the software; PolyglotHIS and other three editions of HIS using individual data-stores. As shown in Figures 7.7,7.8 and 7.9, latency caused due to presence of multiple layers of PolyglotHIS is negligible and merges as the dataset increases. Evidently in terms of functionality, HIS editions involving one data-store can have variance as each data-store supports different range of features. Whereas in case of PolyglotHIS, functionality is limited by the common operations supported by all inherent data-stores. All four versions of HIS were tested for 4 set of loads for retrieval operation only. Taking care of random factors, we have executed each query five times for each of the database. The graphs shown in this section were visualized by calculating the average of the experiments.

Neoclipse is a sub-project undertaken by Neo4j which provides a visualization interface for graphs running in Neo4j instance or can also open other graph files. Neo4j comes with its own browser-based visualizer but Neoclipse provides much more granularity and filters in graph visualization. Neoclipse also provides editing

capabilities besides from visualization and hence has more utility than the default visualizer. It also provides greater insight into every detail of each node, attribute and relationships.

Neoclipse was used for graph visualization, while data stored in Neo4j graph database was accessed through PY2NEO package. Queries of varying complexities were executed on Neo4j alone and then the same queries were executed via the PolyglotHIS system. The total time taken for query execution was recorded for both of the cases. Since the queries dealt only with Neo4j database, hence the total time taken by PolyglotHIS to execute query is basically the sum of time consumed at mediation layer of the PolyglotHIS and the actual time taken to execute the query at data-store layer. Figure 7.7 visualizes the performance comparison of Neo4j with PolyglotHIS for retrieval operation and it shows that query execution time increases rapidly for queries of depth 4. It also portrays that the difference between direct querying Neo4j and querying through multiple layers of PolyglotHIS is not very significant.

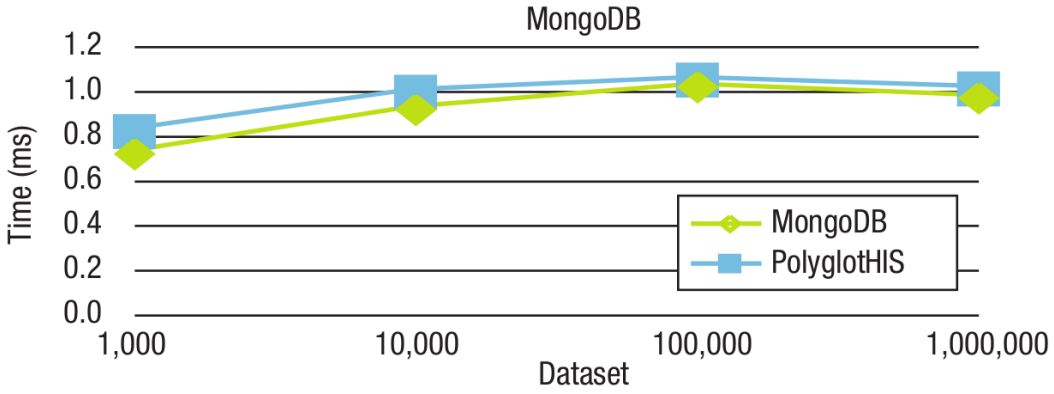

Figure 7.8: Performance comparison of PolyglotHIS with MongoDB for retrieval operation

MongoDB was accessed using mongo shell commands only since there is no popular MongoDB editor available till now. *MongoInspector* was used to extract schema and again pyDatalog was used to store the extracted schema as Datalog facts in

KBoD. MongoClient of PYMONGO package was used for accessing MongoDB using Python programming language. Queries of different complexities were executed on a range of data-set having 1,000 to 1,000,000 documents to show the performance of MongoDB individually as well as of PolyglotHIS. Unlike Neo4j, query execution time of MongoDB remains similar as data-set increase, especially for large data-sets. Figure 7.8 shows the time taken for query execution by MongoDB and PolyglotHIS. Dissimilar to graph shown in Figure 7.7, where the gap between directly querying Neo4j and querying through PolyglotHIS is increasing gradually with increasing depth, in Figure 7.8 this gap is converging as the data-set is increasing. Proving that for larger data-sets, performance of PolyglotHIS is remarkable.

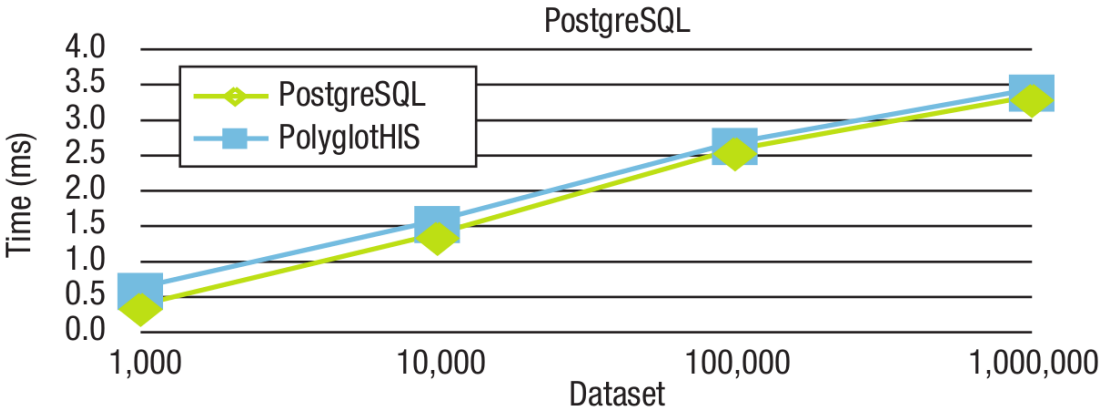

Figure 7.9: Performance comparison of PolyglotHIS with PostgreSQL for retrieval operation

PostgreSQL was used for storing structured data especially that data that demanded conformance of ACID properties. PGADMIN was used for performing administration and development of PostgreSQL. It is a popular open-source graphical user interface for interacting with PostgreSQL. The size of considered databases was increased from 10,000 rows to 1,000,000 rows to be able to show variation in performance of both PostgreSQL and PolyglotHIS as data-set in increased. As shown in Figure 7.9, the query execution time is increasing linearly with the increase in number of records, owing to the difference in implementation of scalability feature.

Relational databases follow vertical scalability, while NoSQL databases follow horizontal scalability Similar to MongoDB performance graph, the difference in query execution time of PostgreSQL and PolyglotHIS merges as data-set increases.

7.4 Conclusion

We have showcased the working of proposed system with the help of an example scenario along with sample data-set. We have also shown example queries of considered NoSQL data-stores in their query languages as they are new and not very well versed in research and developer community. Medical industry has the most unstructured form of data and therefore scenarios dealing with clinical records have been chosen.

Big organizations today, no more rely on one data-storage technology, instead use different databases for different requirements within an application. Managers can easily venture into polyglot-persistent softwares, since all data-stores involved in PolyglotHIS are free and open-source. Undoubtedly, polyglot-persistence is accompanied with over-head of dealing with multiple data-stores, but the benefits are worth every penny.

This chapter demonstrated how the architecture and components of the system were used along with Datalog to plan and execute the queries. Implemented system was evaluated using different queries on increasing data-set to understand how the system behaves as the data-set increases. Evaluation was also done to determine the latency induced by multiple layers of PolyglotHIS in comparison to queries executed directly on their data-stores. Experiments shows that the extra time taken by PolyglotHIS due to presence of mediation layer is negligible. Next chapter concludes the thesis and discusses various challenges and issue faced before and during the implementation of the system. Future scope of this work is also presented which primarily aims at incorporating big data analytics component to the system.

Chapter 8

Implementation Challenges, Conclusion and Future Scope

This chapter discusses various challenges and issues faced during the implementation of the system. Heterogeneity in the underlying data-sources in terms of their data storage models, query syntaxes and data representation mechanisms etc. are the major hindrances for implementation of any data-integration system. These challenges and issues are discussed in detail in this chapter. This chapter also gives concluding remarks on the thesis by highlighting the significant contributions of the research work done.

Polyglot-persistent architectures are increasingly gaining attention in the industry because of the flexibility these systems provide in choosing multiple databases for multiple modules such that overall performance of the system is enhanced.

Future directions of the implemented research work are also presented towards the end of this chapter. Incorporating big data analytics as a component in the architecture of PolyglotHIS is one of the prominent directions for future work, since the data-stores involved are known for their ability to handle big data.

8.1 Challenges and Issues

Design and implementation of Polyglot systems is an intricate task as compared to a traditional system which involves a single database. There are many challenges and issues in the context of these systems, such as '*What will be the data access strategy?*', as involved data-stores are heterogeneous in multiple ways such as storage mechanism, data model, query language etc. This heterogeneity must be invisible to user/application, especially, when a single query requires retrieval of data from more than one data-stores. Another challenging question is how to divide data amongst multiple data-stores to exploit the features offered by each data-store to the greatest extent. While designing a polyglot persistent solution, deciding on appropriate data-store for your storage need is one of the most crucial decisions as well as the most difficult challenge.

Associated with the problem of division is the problem of how to maintain consistency across multiple underlying data-stores. For example, Hospital Information System (HIS) may store data about patients in one data-store and doctors in another. PolyglotHIS should not allow appointment of a patient with a non-existent doctor. Consistency between data-stores is hence very essential to maintain. Finally, there is a challenge to enforce proper rules to synchronize data across the data-stores involved.

Adequately capturing complex capabilities of underlying data-stores and specifying them in the form of a representation that will be used both during query formulation and evaluation is an important research issue. Provision of a layer that hides the details about use of various data models, variety of query languages they use, and their specific schema representation mechanisms is a challenge on the part of the developer. Polyglot persistent softwares should not demand users learn multiple query languages/interfaces to query underlying data.

Firstly, one or more data-stores containing the sought after or relevant data must be identified and then subqueries need to be translated to the native query languages of the required data-stores, thereby necessitating the global query to be decomposed into equivalent set of local queries. After the execution, results so obtained in multiple

formats must be received by the common layer which transforms and combines them before giving the final result back to the application or user. The middle-ware should be responsible for decomposing the query and integrating the results. The software component designated with the task of query syntax conversion from one global query into multiple local queries is given different names in literature, "drivers", "translators","mediators", "wrappers" etc. This whole processes should also consider query optimizations for performance enhancements as well as give due consideration to mappings between the common layer and constituent data-stores. The goal is to provide a consistent database interface that abstracts the heterogeneity of constituent data-stores.

Following are some salient challenges that have been faced during the implementation of PolyglotHIS:

- Minimizing the performance overhead due to query decomposition time, retrieving data from multiple data-stores, result unification time etc.
- Integrating different NoSQL data-stores have different data modeling techniques, specifications and implementations.
- Handling complex queries in the presence of heterogeneous data stores.
- Providing mechanisms for acquiring and representing semantic knowledge and contents of heterogeneous data-stores.
- Handling overlapping fields in data-stores.
- Selection of Data store and generation of efficient query execution plans.
- Achieving transparency for user, which involved:
  - Knowing the schema of member databases.
  - Creating access mechanisms for each data-store.
  - Translating query into the query language of the target database.
- Managing the schema evolution of involved data-stores.

- Designing a common query language for querying NoSQL data-stores.
- Managing variability in query processing performances of each data-store.
- Dealing with contradictory information stored in different data sources.
- Allowing new data-stores to be added to the system, with minimum impact on the existing system.

Scalability of the implemented system in terms of increasing dataset is demonstrated in Chapter 7.1.1 comparing the mediated system against its intrinsic databases. The implemented architecture also allows immediate access to updates in member data-stores and is capable of handling complex queries, while keeping the user unaware of data-store location, access mechanisms and schemas of participating data-stores. Implementation of translators is relatively less difficult than the creation and maintenance of KBoD (Knowledge Base of Data). Also, decision on a uniform query format is very difficult — '*What should be the input and output of translators?*'; Since the translators need to be unified to produce final result, the output of translators should be compatible with each other. In the implemented system, translators accept a query written in the unified query language and generate query in the target query language. Representation and storage of intermediate results produced by queries is made easy with the help of usage of JSON. In this study, all the considered data-stores produce output in JSON format making the unification of the result easier.

8.2 Conclusion

For past few years, relational databases have served as the backbone of the software industry by storing data optimally in an easy-to-understand and structured way. But, relational databases were not designed for handling *Big Data*, and they face challenges and incur performance issues while dealing with un-structured and semi-structured data. Software industry today demands a more realistic model that can

structure data as they are depicted in an application. This has led to the surge of numerous schema-less, highly scalable and dataset-oriented non-relational database systems. This new revolution is named as the NoSQL movement and this movement has witnessed the commencement of such data-stores that do not require a predetermined schema and may very easily capture complex datasets.

Till 2005, database designers had to choose from a very small range of databases that were primarily relational. Recent emergence of schema-less NoSQL databases has opened new avenues for managing data; specifically, the health-care data in our case. With the advent of NoSQL data-stores, database designer's toolbox has significantly increased in number as well as options. More than 200 NoSQL data-stores are available today and almost all of them are open-source. Furthermore, NoSQL data-stores are available under four classes, namely - *the Key-Value, the Document-oriented, the Columnar and the Graph-based* allowing many choices for the developer community. Each NoSQL data-store comes with its own query language and API bindings with the support of various programming languages such as Java, Ruby, Python. Although, learning a query language for each NoSQL data-store is an additional overhead, but at the same time it provides flexibility in querying underlying data in an expressive way. Absence of a uniform query language for posing query to the NoSQL data-stores is one of the biggest hindrances to the quick adoption of NoSQL data-stores.

Modern applications need to store and manipulate data in multiple data-stores repeatedly. However, interacting with heterogeneous data-stores is not an easy task especially when it has to manage complex queries such as Joins. To overcome these problems, researchers have proposed several solutions in order to provide transparent access to heterogeneous data-stores. Some of the proposed solutions are based on the definition of a common API while others bank upon the frameworks that are able to access different data-stores. None of the solutions available in the literature supports Join operation amongst the data-store and also, none of the existing system has worked on the integration of graph data-store with other classes of NoSQL data-stores or relational databases. Supporting Join operation across data-stores is very difficult to implement as it requires management of a temporary data storage

unit. Integration of graph data-stores with other classes is a more complex problem, relative to all other three classes of data-stores, which can be represented in key-value form and hence can be unified based on the key. These two problems, have been successfully solved by the implementation of the framework proposed in this research. A partial result storage component is provided in the proposed architecture to support Join operation, and since we are not representing data present in all data-stores in a common format, to make the support of graph data-stores feasible.

Health-care Information Systems are becoming more complex with the ever increasing demand and high expectations of the large user base. Each department within HIS such as accounting, clinical laboratory, ICU etc. have varying data storage with its own processing requirements. Traditional HIS employ the same data storage model across all the departments resulting in a rigid architecture, limiting data availability, accessibility and usage of their full potential. It has been already shown that NoSQL data-stores outperform relational databases for queries dealing with large data. We propose that instead of forcing all types of data to fit in typical row-column format of relational data-bases, multiple data-stores be used; each storing and representing data in formats closer to their actual representation and usage. Even though different departments in a hospital deal with diverse types of data and deploy distinct data-stores, our framework relieves the end-users from having to interface with each data-store individually by providing a uniform query interface facade.

Provision of a uniform query interface to access the health-care data present in multiple data-stores being used autonomously by various departments of an hospital, empowers the health-care community to reach better decisions and useful conclusions. One approach for implementation of such a system is to use any one of the known database to store and process all the data pertaining to all the departments of a hospital. Nonetheless, any single data-model cannot efficiently store and process multitude of data generated by the healthcare institutions; especially the traditional relational databases are not adequate by themselves. However, modern NoSQL data-stores allow data to be stored in a form closer to their actual representation and usage.

We have developed an intelligent information integration solution named *The Polyglot-persistent Healthcare Information System* (*PolyglotHIS*), which makes use of co-operating agents enabling health-care professionals to retrieve data from heterogeneous data-stores. *PolyglotHIS* uses multiple data-stores for storage and processing of the HIS data. Our framework uses one relational (PostgreSQL) and two NoSQL data-stores namely, the Graph (Neo4j) and the Document (MongoDB). NoSQL data-stores have been used for storing semi-structured data such as medical images and un-structured data such as patient-doctor interaction reports, whereas the relational database is used for storage of structured and financial data. The rationale is to select the most appropriate data storage technology that meets the specific requirements of each module of the HIS. Architecture of PolyglotHIS consists primarily of multiple co-operative agents. Capabilities and contents of data-stores are stored and inferred using Datalog, which is a declarative logic programming language used to store the set of facts and rules. Design and working principle of PolyglotHIS has been illustrated with the help of a running example in Chapter 7. Performance analysis of the implemented system shows that only a small amount of latency has been induced by the system.

The broad goal of PolyglotHIS is to enable access to heterogeneous data-stores, while insulating users from the information pertaining to the location and querying mechanisms of the underlying data-stores. A simple approach is to construct a global schema that captures snapshots of information present in all data-stores, against which user will pose queries. However, this approach is not free from its associated problems–difficulty in integration of multiple and heterogeneous data sources, difficulty in adding new or removing the existing data source and the requirement of reformulation of the global schema to incorporate any updates in the structure of the underlying data-stores. In place of the above mentioned *data translation* approach, a *query translation* mechanism is used in PolyglotHIS, eliminating the need of a global schema. The later has advantages in terms of flexibility, scalability and dynamic query execution. Dynamic query execution takes into account current circumstances such as unavailability of any data-store or addition of a new data-store.

Following steps summarizes the workflow of the proposed framework:

1. Initially, **schema extractor agents** extract schema of respective data-stores and store the fetched information in the form of Datalog facts in KBoD.

   DBAs define Datalog rules, which make use of these Datalog facts to gain more knowledge about the schema of the under-lying data-stores.

2. User poses the query to the system using multiple drop-down menus generated by the **graphical interaction agent**.

3. **Data-store apropos agents** determines the association between the attributes provided by the user and their location in the involved data-stores. This is done with the help of knowledge stored in KBoD which is queried via **Datalog agent**.

4. Data-store apropos agent provides the information obtained in Step 3 to the **query planning agent**, which is responsible for generating possible query plans to retrieve the required data from the data-stores. Query planning agent consults the **learning agent** to ascertain the best possible plan for the query.

   The learning agent is responsible for storing statistics associated with queries executed in past, which helps the query planning agent to make a decision while generating future query plans.

5. The generated query plan is provided to the **query mapping agent** for mapping sub-queries, designated for each data-store, in a particular order as indicated by the query plan.

6. **Translators** convert the sub-queries into query language of respective data-stores.

7. The **partial result storage** is used for storing temporary results, especially in case of join operation.

8. Final compilation of the result and caching the statistics associated with the query in KBoD is done by the **reducer agent**.

9. Final result is displayed to the user with the help of a graphical interaction agent.

Integration of results from three totally different data-stores has been possible with JSON format. Hence, the extensibility of the system is achieved by requiring that any newly added data-store supports JSON format. Another limitation to the extensibility of the system is the non-availability of schema extraction and query translation softwares for the newly added data-store in the existing framework. Query translators are relatively easy to find, than schema extractors. The NoSQL data-stores are schema-less thereby making it difficult to express user needs in a structured query format.

8.3 Future Scope

Dependence on HIS for diagnosis, suggestions, treatments and prescriptions for overall improvement in services and practices is fast increasing. Healthcare institutes have ample amount of data, but there is an acute shortfall of frameworks extracting useful information from this data. Providing a uniform interface to query disparate data stores, when combined with the capability of deriving intelligence from them, will make the system competent, providing a distinctive edge over the existing traditional HIS solutions. Employment of big data analytics in healthcare domain will accelerate efficiency and conjointly will provide breakthroughs in research and stimulate new discoveries [167]. Recently, many researchers have focused on the capabilities of big data analytics in the healthcare sector [168–172].

Some of the future directions of research in the area are:

1. Demonstrating that PolyglotHIS can help the healthcare industry to derive useful information from data by incorporating the provision of deriving intelligence from the stored data.

2. Applying analytics to patient profiles employing strategies such as segmentation and predictive modeling through which the system can pro-actively identify

individuals who may derive benefit of preventative care or lifestyle changes.

The biggest challenge before researchers is for applying analytics or machine learning algorithms, is the limitation that data must be accumulated at one place in an analyzable format. All three data-stores chosen for implementation of the system are scalable and inherently support various data analytics techniques. For example all of them support the combination of Hadoop and map-reduce functionality either directly (MongoDB) or in-directly (PostgreSQL and Neo4j) using wrappers. MongoDB's built-in support for map-reduce model enables the application of data analytics on operational data. Additionally, data stored in *MongoDB* can be used comfortably with analytical tools such as JSON Studio and Pentaho.

3. The researchers plan to use MADlib on *PostgreSQL* for doing predictive analytics, where MADlib is an open-source library to perform in-database analytics. Data stored in *Neo4j* can also be analyzed to discover genetically inherited diseases and to find associations across various doctors/hospitals/laboratories via examining their referral patterns.

4. Spurred by the recent advancements, the researcher plans to make use of popular data-analytics tool – *Tableau* to deduce and visualize useful results from the data. This tool will be very beneficial especially for the doctors enabling them to reach a quick and accurate diagnosis. Tableau has been designed to accelerate real-time conversations with data over multiple data-stores. Hospitals employing Tableau over polyglot-persistence software will benefit from its distinct competitive advantage. Recent versions of Tableau support various NoSQL data-stores, including MongoDB and Neo4j, thereby facilitating big data integration. Features of Tableau such as flexibility and creativity make it the most suitable tool for health-care data analytics.

5. While we have elucidated the applicability of Polyglot-persistence in HIS, considering a variety of data that can easily be segregated to diverse categories of NoSQL data-stores, the concept may be applied equally well to any other

application domains, where different parts of the same application handle different data formats, such as retail applications and social-networking websites. In future it is possible to explore, other use-cases where polyglot-persistence may prove its usefulness.

6. In the current research, we have considered only two classes of NoSQL datastores. The system may be extended to support other classes of databases (key-value, column-oriented, XML, NewSQL) also.

References

[1] Pramod J Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 2012.

[2] Ken Ka-Yin Lee, Wai-Choi Tang, and Kup-Sze Choi. Alternatives to Relational Database: Comparison of NoSQL and XML approaches for Clinical Data Storage. *Computer Methods and Programs in Biomedicine*, 110(1):99–109, 2013.

[3] Robert W Taylor and Randall L Frank. CODASYL Database Management Systems. *ACM Computing Surveys (CSUR)*, 8(1):67–103, 1976.

[4] W. C. McGee. The Information Management System IMS/VS: Part I: General Structure and Operation. *IBM Systems Journal*, 16(2):84–95, June 1977.

[5] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[6] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley publ., 1975.

[7] Edgar F Codd. Does your DBMS run by the Rules? *Computer World*, 21:11, 1985.

[8] Catriel Beeri, Philip A. Bernstein, and Nathan Goodman. A Sophisticate's Introduction to Database Normalization Theory. In *Proceedings of the Fourth International Conference on Very Large Data Bases*, volume 4, pages 113–124. VLDB Endowment, 1978.

REFERENCES

[9] Donald D Chamberlin and Raymond F Boyce. SEQUEL: A Structured English QUEry Language. In *Proceedings of the ACM SIGFIDET (now SIGMOD) Workshop on Data description, Access and Control*, pages 249–264. ACM, 1974.

[10] Arthur M. Keller, Richard Jensen, and Shailesh Agrawal. Persistence Software: Bridging Object-Oriented Programming and Relational Databases. volume 22, pages 523–528. ACM, 1993.

[11] Matthew Aslett. How will the Database Incumbents Respond to NoSQL and NewSQL. *The 451 Group*, pages 1–5, 2011.

[12] Michael Grossniklaus and David Maier. The Curriculum Forecast for Portland: Cloudy with a Chance of Data. *ACM SIGMOD Record*, 41(1):74–77, April 2012.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. 41(6):205–220, 2007.

[15] Stefano Secci and San Murugesan. Cloud Networks: Enhancing Performance and Resiliency. *Computer*, (10):82–85, 2014.

[16] Maged M. Michael, José E. Moreira, Doron Shiloach, and Robert W. Wisniewski. Scale-up x Scale-out: A Case Study using Nutch/Lucene. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007. URL http://www.cecs.uci.edu/~papers/ipdps07/pdfs/SMTPS-201-paper-1.pdf.

REFERENCES

[17] Mario Schkolnick and P Sorenson. The Effects of Denormalization on Database Performance. *Australian Computer Journal*, 14(1):12–18, 1982.

[18] Eric A Brewer. Towards Robust Distributed Systems. In *ACM Symposium on Principles of Distributed Computing*, volume 7, 2000.

[19] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM Special Interest Group on Algorithms and Computation Theory News*, 33(2):51–59, 2002.

[20] Dan Pritchett. BASE: An Acid Alternative. *ACM Queue*, 6(3).

[21] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Googles Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[22] Michael Stonebraker and Ariel Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Engineering Bulletin*, 36(2):21–27, 2013.

[23] Barbara Brynko. NuoDB: Reinventing the Database. *Information Today*, 29 (9):9–9, 2012.

[24] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. Data Management in Cloud Environments: NoSQL and NewSQL Data Stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):22, 2013.

[25] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 2011.

[26] Robin Hecht and Stefan Jablonski. NoSQL Evaluation: A Use Case Oriented Survey. In *International Conference on Cloud and Service Computing*, pages 336–341. IEEE, 2011.

REFERENCES

[27] Jeremy Zawodny. Redis: Lightweight Key/Value Store That Goes the Extra Mile. *Linux Magazine*, August 2009.

[28] Alex Feinberg. Project Voldemort: Reliable Distributed Storage. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, 2011.

[29] Lior Okman, Nurit Gal-Oz, Yaron Gonen, Ehud Gudes, and Jenny Abramov. Security Issues in NoSQL Databases. In *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 541–547. IEEE, 2011.

[30] Tiago Macedo and Fred Oliveira. *Redis Cookbook*. O'Reilly Media, Inc., 2011.

[31] Josiah L Carlson. *Redis in Action*. Manning Publications Co., 2013.

[32] Rob Tweed and George James. A Universal NoSQL Engine, using a tried and tested Technology. *White Paper, Creative Commons Attribution CC-BY*, 2010.

[33] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented Database Systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[34] Prabin Kumar Panigrahi. Business Intelligence at Bharti Airtel Ltd. In *Reshaping Society through Analytics, Collaboration, and Decision Support*, pages 249–266. Springer, 2015.

[35] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. Row-stores: How Different Are They Really? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 967–980, New York, NY, USA, 2008. ACM.

[36] Gordon Rios and Doug Judd. Load Balancing for Hypertable. In *Workshops at the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[37] Lars George. *HBase: The Definitive Guide*. O'Reilly Media, Inc., 2011.

REFERENCES

[38] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[39] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[40] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 671–682, 2006.

[41] David Loshin. Gaining the Performance Edge Using a Column-Oriented Database Management System. *White paper*, 2009. URL http://www.sybase.com/files/White_Papers/Performance-Column-DBM-041509-WP.pdf.

[42] Dhruba Borthakur. The Hadoop Distributed File System: Architecture and Design. *Hadoop Project Website*, 11(2007):21, 2007.

[43] Arnab Pal and Sanjay Agrawal. An Experimental Approach Towards Big-data for Analyzing Memory Utilization on a Hadoop Cluster using HDFS and MapReduce. In *First International Conference on Networks & Soft Computing (ICNSC)*, pages 442–447. IEEE, 2014.

[44] Ankur Khetrapal and Vinay Ganesh. Hbase and hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University*, pages 22–28, 2006.

[45] Sperberg McQueen Tim Bray, Jean Paoli and Eve Maler. Extensible markup language (xml) 1.0 second edition w3c recommendation. *Technical Report RECxml-20001006, World Wide Web Consortium*, October 2000.

[46] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. YAML Ain't Markup Language (YAML) Version 1.1. *Working Draft 2008-05*, 11, 2009.

REFERENCES

[47] Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2013.

[48] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, Inc., 2010.

[49] Brian Ritchie. *RavenDB High Performance*. Packt Publishing Ltd, 2013.

[50] Zhen Hua Liu, Beda Hammerschmidt, and Doug McMahon. JSON Data Management: Supporting Schema-less Development in RDBMS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1247–1258. ACM, 2014.

[51] Peter Membrey, Eelco Plugge, and DUPTim Hawkins. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, 2010.

[52] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. In *Proceedings of the 48th annual Southeast regional conference*, page 42. ACM, 2010.

[53] Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[54] Claudio Tesoriero. *Getting Started with OrientDB*. Packt Publishing Ltd, 2013.

[55] Marko A. Rodriguez and Peter Neubauer. The Graph Traversal Pattern. *CoRR*, abs/1004.1001, 2010.

[56] Justin J Miller. Graph Database Applications and Concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA March 23rd-24th*, 2013.

[57] Peter T. Wood. Query Languages for Graph Databases. *SIGMOD Record*, 41 (1):50–60, 2012.

[58] Florian Holzschuher and René Peinl. Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native access in Neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204. ACM, 2013.

[59] Trevor M. O'Brien, Anna M. Ritz, Benjamin J. Raphael, and David H. Laidlaw. Gremlin: An Interactive Visualization Model for Analyzing Genomic Rearrangements. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):918–926, 2010.

[60] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. *W3C Recommendation*, 4:1–106, 2008. URL http://www.w3.org/TR/rdf-sparql-query/.

[61] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An Open Source Software for Exploring and Manipulating Networks. In Eytan Adar, Matthew Hurst, Tim Finin, Natalie S. Glance, Nicolas Nicolov, and Belle L. Tseng, editors, *ICWSM*, pages 361–362. The AAAI Press.

[62] Maurizio Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 233–246, 2002.

[63] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. An Adaptive Query Execution System for Data Integration. *SIGMOD Record*, 28(2):299–310, June 1999.

[64] Won Kim and Jungyun Seo. Classifying Schematic and Data Heterogeneity in Multidatabase Systems. *Computer*, 24(12):12–18, 1991.

[65] DK Hsiao, EJ Neuhold, and R Sacks-Davis. So far (Schematically) yet so near (Semantically). In *Interoperable Database Systems (DS-5): Proceedings of the IFIP WG2. 6 Database Semantics Conference on Interoperable Database Systems (DS-5) Lorne, Victoria, Australia, 16-20 November, 1992*, page 283. Elsevier, 2014.

[66] Erhard Rahm and Hong Hai Do. Data Cleaning: Problems and Current Approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

[67] Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM computing surveys (CSUR)*, 18(4):323–364, 1986.

[68] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data Integration: The Teenage Years. In *Proceedings of the 32nd international conference on Very Large Data Bases*, pages 9–16. VLDB Endowment, 2006.

[69] Ashish Gupta, Inderpal Singh Mumick, et al. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering bulletin*, 18(2):3–18, 1995.

[70] Stefano Spaccapietra and Christine Parent. View Integration: A Step Forward in Solving Structural Conflicts. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):258–274, 1994.

[71] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. *ACM SIGMOD Record*, 24 (2):316–327, 1995.

[72] Amit P Sheth and James A Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys (CSUR)*, 22(3):183–236, 1990.

[73] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I.* Computer Science Press, Inc., New York, NY, USA, 1988. ISBN 0-88175-188-X.

[74] Edvin A Rusis. Heterogeneous Data Translation System, August 16 1994. US Patent 5,339,434.

[75] Yannis Papakonstantinou, Ashish Gupta, Hector Garcia-Molina, and Jeffrey Ullman. A Query Translation Scheme for Rapid Implementation of Wrappers. In *Deductive and Object-Oriented Databases*, pages 161–186. Springer, 1995.

[76] Serge Abiteboul and Victor Vianu. Procedural and Declarative Database Update Languages. In *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 240–250. ACM, 1988.

[77] Jurgen Annevelink, Charles Y Young, and Paul C Tang. Heterogenous Database Integration in a Physician Workstation. In *Proceedings of the Annual Symposium on Computer Application in Medical Care*, page 368. American Medical Informatics Association, 1991.

[78] Limsoon Wong. Kleisli, a Functional Query System. *Journal of Functional Programming*, 10(1):19–56, January 2000.

[79] John Y Park and Mark A Musen. Mappings for Reuse in Knowledge-based Systems. In *Industrial Knowledge Management*, pages 349–363. Springer, 2001.

[80] Walter Sujansky and Russ Altman. Bridging the Representational Heterogeneity of Clinical Databases. In *Spring Symposium: Artificial Intelligence in Medicine*, 1994.

[81] Yuri Breitbart. Multidatabase Interoperability. *ACM SIGMOD Record*, 19(3): 53–60, 1990.

[82] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. Overview of Multidatabase Transaction Management. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research-Volume 2*, pages 23–56. IBM Press, 1992.

[83] Bogdan D Czejdo, Marek Rusinkiewicz, and David W Embley. An Approach to Schema Integration and Query Formulation in Federated Database Systems. In

Proceedings of the Third International Conference on Data Engineering, pages 477–484. IEEE Computer Society, 1987.

[84] Gio Wiederhold. Mediation to Deal with Heterogeneous Data Sources. In *Interoperating Geographic Information Systems*, pages 1–16. Springer, 1999.

[85] Alon Y. Halevy. Answering Queries Using Views: A Survey. *The VLDB Journal*, 10(4):270–294, December 2001.

[86] Erhard Rahm and Philip A Bernstein. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10(4):334–350, 2001.

[87] Ronald Fagin, Phokion G Kolaitis, Renée J Miller, and Lucian Popa. Data Exchange: Semantics and Query Answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[88] Marc Friedman, Alon Y Levy, Todd D Millstein, et al. Navigational Plans for Data Integration. *Innovative Applications of Artificial Intelligence Conference*, 1999:67–73, 1999.

[89] Alon Y. Levy. The Information Manifold Approach to Data Integration. *IEEE Intelligent Systems*, 13:12–16, 1998.

[90] Michael R Genesereth, Arthur M Keller, and Oliver M Duschka. Infomaster: An Information Integration System. In *ACM SIGMOD Record*, volume 26, pages 539–542. ACM, 1997.

[91] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey Ullman, Vasilis Vassalos, and Jennifer Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of intelligent information systems*, 8(2):117–132, 1997.

[92] Erik M van Mulligen, Teun Timmers, Jaap Brand, Ronald Cornet, Freek van den Heuvel, Martin Kalshoven, and Jan H van Bemmel. HERMES: A Health Care Workstation Integration Architecture. *International Journal of Bio-medical Computing*, 34(1):267–275, 1994.

[93] M.J. Carey, L.M. Haas, P.M. Schwarz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, A.W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J.H. Williams, and E.L. Wimmers. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Fifth International Workshop on Research Issues in Data Engineering*, pages 124–131, Mar 1995.

[94] Case Squire. Data Extraction and Transformation for the Data Warehouse. *SIGMOD Record*, 24(2):446–447, May 1995.

[95] Jennifer Widom. Research Problems in Data Warehousing. In *Proceedings of the Fourth International Conference on Information and Knowledge Management*, pages 25–30. ACM, 1995.

[96] Thomas R Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[97] Yigal Arens, Chun-Nan Hsu, and Craig A Knoblock. Query Processing in the SIMS Information Mediator. *Advanced Planning Technology*, 32:78–93, 1996.

[98] Holger Wache, Thomas Voegele, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, Holger Neumann, and Sebastian Hübner. Ontology-based Integration of Information - A Survey of Existing Approaches. In *IJCAI-01 Workshop: Ontologies and Information Sharing*, volume 2001, pages 108–117, 2001.

[99] Douglas Lenat, Michael Witbrock, David Baxter, Eugene Blackstone, Chris Deaton, Dave Schneider, Jerry Scott, and Blake Shepard. Harnessing Cyc to Answer Clinical Researchers' Ad-hoc Queries. *Artificial Intelligence Magazine*, 31(3):13–32, 2010.

[100] Eduardo Mena, Arantza Illarramendi, Vipul Kashyap, and Amit P Sheth. OBSERVER: An Approach for Query Processing in Global Information Systems based on Interoperation Across Pre-existing Ontologies. *Distributed and Parallel Databases*, 8(2):223–271, 2000.

[101] Gene Golovchinsky. What the Query told the Link: The Integration of Hypertext and Information Retrieval. In *Proceedings of the eighth ACM conference on Hypertext*, pages 67–74. ACM, 1997.

[102] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform access to non-relational database systems: The SOS platform. In *Advanced Information Systems Engineering*, pages 160–174. Springer, 2012.

[103] Luca Cabibbo. ONDM: An Object-NoSQL Datastore Mapper. *Faculty of Engineering, Roma Tre University*, 2013.

[104] Francesca Bugiotti, Luca Cabibbo, Paolo Atzeni, and Riccardo Torlone. A Logical Approach to NoSQL Databases , 2013.

[105] Olivier Curé, Robin Hecht, Chan Le Duc, and Myriam Lamolle. Data Integration over NoSQL Stores Using Access Path Based Mappings. In *Database and Expert Systems Applications*, pages 481–495. Springer, 2011.

[106] Olivier Curé, Fadhela Kerdjoudj, Chan Le Duc, M Lamolle, and David Faye. On The Potential Integration of an Ontology-Based Data Access Approach in NoSQL Stores. In *Third International Conference on Emerging Intelligent Data and Web Technologies*, pages 166–173. IEEE, 2012.

[107] John Roijackers and George H. L. Fletcher. On Bridging Relational and Document-Centric Data Stores. In *Big Data - 29th British National Conference on Databases*, pages 135–148, 2013.

[108] Roger Lawrence. Integration and Virtualization of Relational SQL and NoSQL Systems including MySQL and MongoDB. In *International Conference on Computational Science and Computational Intelligence (CSCI)*, volume 1, pages 285–290. IEEE, 2014.

[109] Rami Sellami, Sami Bhiri, and Bruno Defude. ODBAPI: A Unified REST API for Relational and NoSQL Data Stores. In *IEEE International Congress on Big Data (BigData Congress)*, pages 653–660. IEEE, 2014.

[110] Ebtesam Alomari, Ahmed Barnawi, and Sherif Sakr. CDPort: A Portability Framework for NoSQL Datastores. *Arabian Journal for Science and Engineering*, pages 1–23, 2015.

[111] Dana Petcu, Beniamino Di Martino, Salvatore Venticinque, Massimiliano Rak, Tamás Máhr, Gorka Esnal Lopez, Fabrice Brito, Roberto Cossu, Miha Stopar, Svatopluk Šperka, et al. Experiences in building a mOSAIC of clouds. *Journal of Cloud Computing*, 2(1):1–22, 2013.

[112] Eleni Kamateri, Nikolaos Loutas, Dimitris Zeginis, James Ahtes, Francesco DAndria, Stefano Bocconi, Panagiotis Gouvas, Giannis Ledakis, Franco Ravagli, Oleksandr Lobunets, et al. Cloud4SOA: A Semantic Interoperability PAAS Solution for Multi-cloud Platform Management and Portability. In *Service-Oriented and Cloud Computing*, pages 64–78. Springer, 2013.

[113] Danilo Ardagna, Elisabetta Di Nitto, Giuliano Casale, Dana Petcu, Parastoo Mohagheghi, Sébastien Mosser, Peter Matthews, Anke Gericke, Cyril Ballagny, Francesco D'Andria, et al. Modaclouds: A Model-driven Approach for the Design and Execution of Applications on Multiple Clouds. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, pages 50–56. IEEE Press, 2012.

[114] Karamjit Kaur and Rinkle Rani. Managing Data in Healthcare Information Systems: Many Models, One Solution. *Computer*, (3):52–59, 2015.

[115] Henrique Valer, Caetano Sauer, and Theo Härder. XQuery Processing over NoSQL Stores. In *Grundlagen von Datenbanken*, pages 75–80, 2013.

[116] Dag Olav Prestegarden. UnQL: A Query Language for NoSQL Databases. 2012.

[117] Daniela Florescu and Ghislain Fourny. JSONiq: The history of a query language. *IEEE Internet Computing*, (5):86–90, 2013.

[118] Debashish Ghosh. Multiparadigm Data Storage for Enterprise Applications. *IEEE Software*, 27(5):57–60, Sept 2010.

[119] Walter Sujansky. Heterogeneous Database Integration in Biomedicine. *Journal of biomedical informatics*, 34(4):285–298, 2001.

[120] Wen-Syan Li, Jianfeng Yan, Ying Yan, and Jin Zhang. Xbase: Cloud-enabled Information Appliance for Healthcare. In *EDBT*, pages 675–680, 2010.

[121] Sanjoy Singh Ningthoujam, Manabendra Dutta Choudhury, Kumar Singh Potsangbam, Pankaj Chetia, Lutfun Nahar, Satyajit D Sarker, Norazah Basar, and Anupam Das Talukdar. NoSQL Data Model for Semi-automatic Integration of Ethnomedicinal Plant Data from Multiple Sources. *Phytochemical Analysis*, 25(6):495–507, 2014.

[122] Yang Jin, Tang Deyu, and Zhou Yi. A Distributed Storage Model for EHR Based on HBase. In *International Conference on Information Management, Innovation Management and Industrial Engineering (ICIII)*, volume 2, pages 369–372, Nov 2011.

[123] Baraa Mohamad, Laurent d'Orazio, and Le Gruenwald. Towards a Hybrid Row-Column Database for a Cloud-based Medical Data Management System. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, page 2. ACM, 2012.

[124] Lihong Jiang, Li Da Xu, Hongming Cai, Zuhai Jiang, Fenglin Bu, and Boyi Xu. An IoT-oriented Data Storage Framework in Cloud Computing Platform. *IEEE Transactions on Industrial Informatics*, 10(2):1443–1451, 2014.

[125] Luca Lianas, Francesca Frexia, Giovanni Delussu, Paolo Anedda, and Gianluigi Zanetti. pyEHR: A Scalable Clinical Data Management Toolkit for Biomedical Research Projects. In *16th International Conference on e-Health Networking, Applications and Services (Healthcom)*, pages 370–374. IEEE, 2014.

[126] Michael Stonebraker. Stonebraker on NoSQL and Enterprises. *Communications of the ACM*, 54(8):10–11, 2011.

[127] Martyn Ellison, Radu Calinescu, and Richard Paige. Re-engineering the Database Layer of Legacy Applications for Scalable Cloud Deployment. In *IEEE 7th International Conference on Utility and Cloud Computing (UCC)*, pages 976–979. IEEE, 2014.

[128] Genoveva Vargas-Solar. Addressing data management on the cloud: Tackling the big data challenges. In *Proceedings of the 7th Alberto Mendelzon International Workshop on Fundations of Data Management, Puebla/Cholula, Mexico, May 21-23*, 2013.

[129] Ian Gorton, John Klein, and Albert Nurgaliev. Architecture Knowledge for Evaluating Scalable Databases. Technical report, Carnegie Mellon University Pittsburgh, Software Engineering Institute, DTIC Document, 2015.

[130] Deka Ganesh Chandra. BASE analysis of NoSQL database. *Future Generation Computer Systems*, 52:13–21, 2015.

[131] Adam Marcus. The NoSQL Ecosystem. *The Architecture of Open Source Applications*, pages 185–205, 2011.

[132] C Mohan. History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 11–16. ACM, 2013.

[133] Cory Nance, Travis Losser, Reenu Iype, and Gary Harmon. Nosql vs RDBMS - Why There is Room for Both. In *Proceedings of the Southern Association for Information Systems Conference*, pages 111–116, 2013.

[134] Juan Castrejón, Genoveva Vargas-Solar, Christine Collet, and Rafael Lozano. Model-driven cloud data storage. *Proceedings of CloudMe*, 2012.

[135] Genoveva Vargas-Solar. Polyglot Persistence and Multi-cloud Data Management Solutions. In *Tutorial Talk in EDBT Summer School on Data all around Big, Linked, Open*, 2013.

[136] Srikrishna Prasad and SB Avinash. Application of polyglot persistence to enhance performance of the energy data management systems. In *International Conference on Advances in Electronics, Computers and Communications (ICAECC)*, pages 1–6. IEEE, 2014.

[137] Santasriya Prasad and MS Sha. NextGen Data Persistence Pattern in Healthcare: Polyglot Persistence. In *Computing, Communications and Networking Technologies (ICCCNT), 2013 Fourth International Conference on*, pages 1–8. IEEE, 2013.

[138] Genoveva Vargas-Solar and José-Luis Zechinelli-Martini. Moving Energy Consumption Control into the Cloud by Coordinating Services. *International Journal of Computers and Their Applications*, page 236, 2013.

[139] Stephanie Fan. An Exploration of Finding Aid Technologies and NoSQL Databases. *See Also*, 1(1), 2015.

[140] Nikhil Mangle and Praful B Sambhare. A Review on Big Data Management and NoSQL Databases in Digital Forensics.

[141] Abhinav Tiwari. Hybrid Data Storage Framework for the Biometrics Domain. 2014.

[142] Pouria Amirian, Anahid Basiri, and Adam Winstanley. Evaluation of Data Management Systems for Geospatial Big Data. In *Computational Science and Its Applications–ICCSA 2014*, pages 678–690. Springer, 2014.

[143] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, Angela Hung Byers, and McKinsey Global Institute. Big data: The Next Frontier for Innovation, Competition, and Productivity. 2011.

[144] Venkat N Gudivada, Dantam Rao, and Vijay V Raghavan. NoSQL Systems for Big Data Management. In *IEEE World Congress on Services*, pages 190–197. IEEE, 2014.

[145] Mehmet Zahid Ercan and Michael Lane. An Evaluation of NoSQL Databases for EHR Systems. In *Proceedings of the 25th Australasian Conference on Information Systems*. Auckland University of Technology, School of Business Information Systems, 2014.

[146] Wei Xu, Zhonghua Zhou, Hong Zhou, Wu Zhang, and Jiang Xie. MongoDB Improves Big Data Analysis Performance on Electric Health Record System. In *Life System Modeling and Simulation*, volume 461 of *Communications in Computer and Information Science*, pages 350–357. Springer, 2014. ISBN 978-3-662-45282-0. doi: 10.1007/978-3-662-45283-7_36.

[147] Tracy D Gunter and Nicolas P Terry. The Emergence of National Electronic Health Record Architectures in the United States and Australia: Models, Costs, and Questions. *Journal of Medical Internet Research*, 7(1), 2005.

[148] John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe, Kim Pham, and Chrisjan Matser. Performance Evaluation of NoSQL Databases: A Case Study. In *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*, pages 5–10. ACM, 2015.

[149] Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. Query Reformulation for Dynamic Information Integration. *Journal of Intelligent Information Systems*, 6(2-3):99–130, August 1996.

[150] Michael R Genesereth and Steven P Ketchpel. Software Agents. *Communications of ACM*, 37(7):48–53, 1994.

[151] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. KQML as an Agent Communication Language. In *Proceedings of the Third International Conference on Information and Knowledge Management*, pages 456–463. ACM, 1994.

[152] R. J. Bayardo, Jr., W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. InfoSleuth: Agent-based Semantic Integration of Information in Open and Dynamic Environments. *SIGMOD Record*, 26(2):195–206, 1997. ISSN 0163-5808. doi: 10.1145/253262.253294.

[153] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What You Always Wanted to Know About Datalog (and Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[154] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and Emerging Applications: An Interactive Tutorial. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1213–1216. ACM, 2011.

[155] William F Clocksin, Christopher S Mellish, and WF Clocksin. *Programming in PROLOG*, volume 5. Springer, 1987.

[156] Oliver M Duschka, Michael R Genesereth, and Alon Y Levy. Recursive Query Plans for Data Integration. *The Journal of Logic Programming*, 43(1):49–73, 2000.

[157] Ronald Fagin, Laura M Haas, Mauricio Hernández, Renée J Miller, Lucian Popa, and Yannis Velegrakis. CLIO: Schema Mapping Creation and Data Exchange. In *Conceptual Modeling: Foundations and Applications*, pages 198–236. Springer, 2009.

[158] Zachary G Ives, Nitin Khandelwal, Aneesh Kapur, and Murat Cakir. ORCHESTRA: Rapid, Collaborative Sharing of Dynamic Data. In *CIDR*, pages 107–118, 2005.

[159] Balder ten Cate and Phokion G Kolaitis. Schema Mappings: A Case of Logical Dynamics in Database Theory. In *Johan van Benthem on Logic and Information Dynamics*, pages 67–100. Springer, 2014.

[160] Raymond M Smullyan. *First-order Logic*. Courier Corporation, 1995.

[161] Raghu Ramakrishnan and Jeffrey D Ullman. A Survey of Deductive Database Systems. *The Journal of Logic Programming*, 23(2):125–149, 1995.

[162] Rajasekar Krishnamurthy, Raghav Kaushik, and Jeffrey F Naughton. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. In *Database and XML Technologies*, pages 1–18. Springer, 2003.

[163] Veronika Abramova and Jorge Bernardino. NoSQL Databases: MongoDB vs cassandra. In *Proceedings of the International Conference on Computer Science and Software Engineering*, pages 14–22. ACM, 2013.

[164] Karamjit Kaur and Rinkle Rani. Modeling and Querying Data in NoSQL Databases. In *IEEE International Conference on Big Data*, pages 1–7. IEEE, 2013.

[165] Renzo Angles. A Comparison of Current Graph Database Models. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 171–177. IEEE, 2012.

[166] Mark Lutz. *Programming Python*. O'Reilly, 1996.

[167] Wullianallur Raghupathi and Viju Raghupathi. Big data analytics in healthcare: Promise and potential. *Health Information Science and Systems*, 2(1):3, 2014.

[168] Yichuan Wang, LeeAnn Kung, Chaochi Ting, and Terry Anthony Byrd. Beyond a Technical Perspective: Understanding Big Data Capabilities in Health Care. In *Proceedings of 48th Annual Hawaii International Conference on System Sciences (HICSS), Kauai, Hawaii*, 2015.

[169] Mohammad Daneshvar Kakhki, Rahul Singh, and Kathy White Loyd. Developing Health Analytics Design Artifact for Improved Patient Activation: An

On-going Case Study. In *New Contributions in Information Systems and Technologies*, volume 353 of *Advances in Intelligent Systems and Computing*, pages 733–739. Springer. doi: 10.1007/978-3-319-16486-1_72.

[170] Ruben Amarasingham, Rachel E Patzer, Marco Huesch, Nam Q Nguyen, and Bin Xie. Implementing Electronic Health Care Predictive Analytics: Considerations and Challenges. *Health Affairs*, 33(7):1148–1154, 2014.

[171] Matthew Herland, Taghi M Khoshgoftaar, and Randall Wald. A Review of Data Mining using Big Data in Health Informatics. *Journal Of Big Data*, 1 (1):2, 2014.

[172] Seth Earley. The Promise of Healthcare Analytics. *IT Professional*, (2):7–9, 2015.

List of Publications

International Journals (indexed by SCI)

1. Karamjit Kaur and Rinkle Rani, "Managing Data in Healthcare Information Systems: Many Models, One Solution", Computer, vol. 48, no. 3, pp. 52-59, March 2015, doi:10.1109/MC.2015.77, *(**Published** by IEEE Computer Society, Thomson Reuter's Journal Citation Index, Impact Factor: 1.438, ISSN: 0018-9162)*

2. Karamjit Kaur and Rinkle Rani, "Smart Polyglot Solution for Healthcare Big Data", IT Professional, *(**Accepted**, IEEE Computer Society, Thomson Reuter's Journal Citation Index, Impact Factor: 0.819,ISSN: 1520-9202)*

International Conferences

1. Karamjit Kaur and Rinkle Rani, "Collaborating Heterogeneous Data-stores in Health-care Information Systems", 10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, Miami, Florida, **USA** 22-25 October 2014.

2. Karamjit Kaur and Rinkle Rani "Moving from Relational to Non-relational Databases", International Conference of Women Engineers and Scientists, Los Angeles, California, **USA**, 23-25 October 2014.

3. Karamjit Kaur and Rinkle Rani, "Polyglot Joins in Cloud", Grace Hopper Conference of Women in Computing, Phoenix, Arizona, **USA**, 8-10 October 2014.

4. Karamjit Kaur and Rinkle Rani, "Modelling and Querying Data in NoSQL Databases", IEEE International Conference on BigData, Silicon Valley San Francisco, **USA**, 6-9 Oct 2013.

5. Karamjit Kaur and Rinkle Rani, "Integration of Relational and Non Relational Databases", Grace Hopper Conference of Women in Computing, Minneapolis, Minnesota, **USA**, 1-5 Oct 2013.

6. Karamjit Kaur Cheema and Rinkle Rani, "A Comprehensive Study of classification of NoSQL Databases", International Conference on Advancement in Computing and Communication (ICACC–2012), 23-25 Feb 2012 at BBSBEC, Fatehgarh Sahib, Punjab, **India**.

Symposium and School

1. Attended EDBT Summer School on *Data all around Big, Linked, Open* from 1-6 September 2013 held at Aussois, France.

2. Attended Public Health Symposium on *Health Informatics: Opportunities and Challenges* from 7-8 March 2015 held at PGIMER, Chandigarh, India.

Paper Communicated

1. Karamjit Kaur, Rinkle Rani,"Realization of Multi-model Query Facade using Co-operative agents and Datalog", ACM SIGMOD Record, *(**Communicated**, SCI-E, Impact Factor: 0.96)*

2. Karamjit Kaur, Rinkle Rani,"Smart Integration of Healthcare Big-Data", The Journal of Supercomputing, *(**Communicated**, SCI, Impact Factor: 0.841)*

Scholarships Awarded

1. Awarded Scholarship by **ThoughtWorks** for presenting a research paper titled "Polyglot Joins in Cloud" in Grace Hopper Conference of Women in Computing, Phoenix, Arizona, **USA** 8-10 October 2014 which included travel, registration and accommodation.

2. Awarded **Google Global Scholarship** for presenting a research paper titled "Integration of Relational and Non Relational Databases" in Grace Hopper Conference of Women in Computing, Minneapolis, Minnesota, **USA** 1-5 Oct 2013 which included travel, registration and accommodation.

3. Awarded full financial assistance from Department of Science and Technology (DST), India to attend one week EDBT (Extended DataBase Technology) Summer School held in Aussois, **France** during 1-6 September 2013.