A COMPARATIVE STUDY OF DIFFERENT DATABASE TECHNOLOGIES FOR BIG DATA MODELING AND ANALYSIS IN EDUCATION

ÖZKAN SAYIN

JANUARY 2015

A COMPARATIVE STUDY OF DIFFERENT DATABASE TECHNOLOGIES FOR BIG DATA MODELING AND ANALYSIS IN EDUCATION

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF

NATURAL AND APPLIED SCIENCES OF

IZMIR UNIVERSITY OF ECONOMICS

BY

ÖZKAN SAYIN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

IN THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

JANUARY 2015

M.S. THESIS EXAMINATION RESULT FORM

Approval of the Graduate School of Natural and Applied Sciences

[illegible]

Prof. Dr. Cüneyt Güzeliş
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

[illegible]

Prof. Dr. Turhan Tunalı
Head of Department

We have read the thesis entitled **"A Comparative Study of Different Database Technologies for Big Data Modeling and Analysis in Education"** completed by **ÖZKAN SAYIN** under supervision of **Prof. Dr. Brahim Hnich** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Prof. Dr. Brahim Hnich
Supervisor

Examining Committee Members **Date:** 22.01.2015

Prof. Dr. Brahim Hnich Dept. of Comp. Sci., IUE	[illegible]
Assoc. Prof. Dr. Diaa Gadelmavla Dept. of Electrical and Electronics Eng., IUE	[illegible]
Asst. Prof. Dr. Burkay Genç Inst. of Population Studies, Hacettepe Univ.	[illegible]

ABSTRACT

A COMPARATIVE STUDY OF DIFFERENT DATABASE TECHNOLOGIES FOR BIG DATA MODELING AND ANALYSIS IN EDUCATION

ÖZKAN SAYIN
M.S. in Computer Science
Graduate School of Natural and Applied Sciences
Supervisor: Prof. Dr. Brahim Hnich
January 2015

With the increase in data generation, notion of Big Data emerged, along with new problems on the side. Traditional relational databases on single computers failed to perform at required efficiencies. As a result, new approaches to hosting data emerged that uses clouds of commodity hardware. In addition, new database management system (DBMS) technologies are created under NoSQL movement, with new ways of modelling data.

Different data models have their own advantages and disadvantages. Consequently, there is not one DBMS that is the best choice for every project. Instead, the way the project needs data to be stored and retrieved is a determinant factor on the choice. Some data models ensure data consistency and ease maintenance; whereas, others focus on performance. We analyse three different data models, namely relational, document based and graph databases, and conduct a case study on Sınavo, an online education system. We investigate each data model from their design to their performances on different queries. We show that different systems offer different qualities and perform better at some queries and worse on others.

In addition to storing Big Data, making data-driven decisions is an important and valuable process. We investigate two exemplary cases on Sınavo. We introduce a novel approach to estimating student performances by applying bayesian statistics on data stored in Sınavo system. We also propose a way of classifying questions based on their difficulty levels.

Keywords: big data, database, data model, sql, relational database, document store, document based database, graph database, data driven decision-making.

ÖZ

BÜYÜK VERİ MODELLEME İÇİN FARKLI VERİTABANI SİSTEMLERİ VE EĞİTİM SİSTEMLERİNDE ANALİZ ÜZERİNE KARŞILAŞTIRMALI BİR ARAŞTIRMA

ÖZKAN SAYIN
Bilgisayar Bilimleri, Yüksek Lisans
Fen Bilimleri Enstitüsü
Tez Danışmanı: Prof. Dr. Brahim Hnich
Ocak 2015

Veri yaratım hızındaki artış ile Büyük Veri kavramı, yanında birçok yeni sorun ile ortaya çıktı. Tek bilgisayar üzerinde çalışan geleneksel ilişkisel veritabanları, istenen verimliliği sağlayamamaya başladı. Sonuç olarak, bu veriyi bilgisayar bulutlarında saklayan yeni yaklaşımlar, ve veriyi farklı şekillerde modelleyen yeni veritabanı yönetim teknolojileri geliştirildi.

Farklı veri modelleri farklı avantajlar ve dezavantajlar sunmaktadır. Bu nedenle, tüm projeler için en iyisi olan bir veritabanı yönetim sistemi yoktur. Aksine, bir proje için doğru olan veritabanı sistemi, bu projedeki verinin nasıl depolanacağına ve sorgulanacağına bağlıdır. Kimi veri modelleri bakımı kolaylaştırır ve veri tutarlılığını garanti altına alırken, kimi verimliliğe odaklanmaktadır. Bu tezde, üç farklı (ilişkisel, döküman tabanlı ve grafik tabanlı) veritabanı sistemi incelenmiş, ve bir çevrimiçi eğitim sistemi olan Sınavo üzerinde örnek vaka çalışması yapılmıştır. Bu üç veritabanı sistemi, tasarım aşamasından, sorgu verimliliğine kadar incelenmiştir. Sonuç olarak, farklı veritabanı sistemlerinin farklı avantajlar sağladığı, ve farklı sorgu tiplerine göre değişik performans sergilediği gösterilmiştir.

Büyük Veriyi depolama ve sorgulamanın dışında, veri güdümlü karar verme çok önemli ve değerli bir işlemdir. Bu tezde, örnek olarak 2 durum incelenmiştir. Sınavo sisteminde öğrencilerin performanslarını sistemin depoladığı istatistikler üzerinden bayes metodlarını kullanarak tahmin etmek için yeni bir yol önerilmiş, ayrıca soruları zorluklarına göre gruplandırmak için bir metot gösterilmiştir.

Anahtar Kelimeler: büyük veri, veritabanı, veri modeli, sql, ilişkisel veritabanı, döküman tabanlı veritabanı, grafik veritabanı, veri güdümlü karar verme.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Topic of the thesis

Everyday life is getting swarmed with computers. From home automations to means of communication, and different kinds of sensors, data generation rate is growing wildly. Although computer systems have been around for decades, with new uses of these systems, such as social networks, the amount of data generated and processed is becoming a problem on its own. Such collections of data are regarded as Big Data.

Database management systems (DBMS) are special software applications that are used to provide storage and means of retrieval for organized collections of data, i.e. databases. In traditional systems with relatively small datasets, data have been stored in relational databases as tables. There are a number of DBMS's that implement relational schemas, such as Microsoft SQL Server, or MySQL, generally characterized with their shared language of querying, SQL (Structured Query Language). However, with new means of data generation, characteristics of data are changed. Social networks, for example, generate highly connected data. In addition, the ways of querying these collections have also evolved. Traditionally, queries included minimal joins of different collections, such as matching an entity in one collection to another entity in another collection. However, social networks now support various types of queries. Users can see not only their friends, but also friends of their friends. They can even check which of these people work for a certain company, or follow certain football team. Sınavo is an online education system that is a good example for the shift in data generation and querying requirements. It provides students with means of solving questions within different context such as games or tests, free of charge. In addition, the system is designed to support socialization among students by sharing questions and their thoughts on them, along with chatting during competitive games.

With new requirements of extracting data from databases, traditional DBMS's began to fail operating effectively. To respond such requirements, new data models for databases emerged. There are many database management systems that are built around these new data models, generally characterized as NoSQL, which is often interpreted as "not only SQL", instead of "no to SQL". Each database management system usually implements its own query language.

Some of these systems focus on new data models, such as storing data as objects, or graph; whereas, others focus on storing and serving any data effectively on clouds of computers, such as Apache's Cassandra. This thesis focuses on ways of modelling data, instead of efficiency in using clouds of computers. As there are tens of data models, such as graph databases, key-value stores, column-based databases, and multiple database management systems for each model exist, it is not within the scope of this thesis to investigate all models and technologies. Instead, we focus on specific and characteristic data models, (document and graph) and report a comparison among them to support our claim.

In addition, we present a novel way of making data-driven decisions regarding student performances and question classifications based on statistics gathered by Sınavo system over time.

1.2 Overview of this thesis

This thesis discusses new problems that arise with Big Data and how different approaches to storing and processing Big Data work. We investigate details of three database management systems of different models: Microsoft SQL Server for relational databases, MongoDB for document stores, and Neo4j for graph databases. We give exemplary systems of different data models for the same problem and discuss what each approach thrives and fails at. We explore bases for database selection and discuss how requirements govern data model.

We also present a novel way of estimating student performances. We make use of bayesian statistics to evaluate student statistics and generate credible intervals which denote probable success rate for students. We discuss how student performances change over time and how a feedback loop within this evaluation system can be used to adjust accordingly. In addition, we apply a similar approach to classification of questions. Again using bayesian statistics, we give a process of data-driven decision making to come up with question difficulties based on statistics gathered over time.

Contributions of this thesis can be summarized as follows:

- We discuss three database systems and details of querying them. We present a case study on Sınavo and how data can be modelled for each database system.
- We provide a comparative analysis of these three database management systems' non-quantitative properties such as maintenance or flexibility.
- We present results of experiments run on three databases and their performances for different types of queries.
- We present a novel way to use statistics stored by Sınavo to make data-driven estimations on student performances by using bayesian credible intervals.

- We present a novel way of classifying questions in an educational system based on their difficulties by using credible intervals. We introduce an approach to use their statistics and map questions to different difficulty labels.

1.3 Structure of the thesis

Following chapters of this thesis are structured as follows:

- Chapter 2 presents the definition of Big Data and exemplifies possible sources and uses. It also investigates fundamental approaches to both storing and processing Big Data.
- Chapter 3 discusses main database models and explains their inner workings. For each database model, we give examples of how data is stored, retrieved and updated.
- Chapter 4 explains what Sınavo is and how it is used by students. We also give statistics of Sınavo to explain why it is a Big Data project.
- Chapter 5 exemplifies different data models applied to Sınavo and give details of how different database management systems are implemented. We also give a comparison among those DMBS's for their quantitative and non-quantitative properties.
- Chapter 6 presents a novel way of data-driven decision making regarding user performances and question difficulties based on statistics gathered by Sınavo system.
- Chapter 7 summarizes this thesis and discusses future work.

Chapter 2

Big Data

The term “Big Data” is used for collections of large and complex datasets whose size is too huge to be operated by traditional computational methodologies. These operations include but are not limited to storing the data, searching the dataset within feasible time and performing analysis to derive useful information.

2.1 Introduction

We live in information age, in which almost every aspect of our lives is digitalised. We have satellites taking images of the world into tiny bits of details. We have weather balloons equipped with sensors to gather meteorological information to be used for forecasting. We shop online, and communicate via Internet. We even have applications on our mobile phones to track our movements as we sleep that try to wake us up when our sleep is light and assess our sleep quality. As a result, we are bombarded with information the amount of which is not easy to comprehend. According to a study done by IBM, 2.5 quintillion (2.5×10^{18}) bytes of data is generated every day. 90% of the data we currently have has been generated in the last two years, and things are not slowing down. McKinsey Global Institutes projects 40% growth in the data generation per year.

Even though the term Big Data brings firstly the size aspect to mind, the amount is not the only dimension that makes a dataset a Big Data. There is a popular $3V$ approach that is used to describe main characteristics of a big dataset, which are *volume* (the amount of the data), *velocity* (the pace of the data generation) and *variety* (the complexity of the data). Note that any or all of these characteristics can be enough to make it as Big Data.

Velocity and volume characteristics of a dataset are easy to measure and are tightly coupled, as faster data generation results in more data generated. Variety, on the other hand, is harder to measure, usually referring to two aspects of data: being heterogeneous, i.e. including different types such as images, texts, arrays of numbers, and being unstructured, i.e. not having a common structure for different samples. Heterogeneity and the lack of structure is mostly caused by different sources that feed the system with data.

2.2 Sources and Uses

There are practically infinite kinds of sources for data, as the world has become embedded with computer systems. Hence, we are going to discuss major types of sources and benefits of analysing data gathered from these sources.

2.2.1 Sensors and Surveillance Systems

Our lives are being monitored. There are many types of sensors, which can be described as tools of digitizing physical entities, scattered all around the world. Planes, for example, record gigabytes of data per flight, which are used to take precautionary measures to prevent risks of crashes and to discover weak points and improve the overall designs.

Surveillance systems installed within urban territories gather streams of videos to help the law-enforcement for monitoring public areas and solving criminal investigations. These systems are called Close-Circuit TV (CCTV), as they record

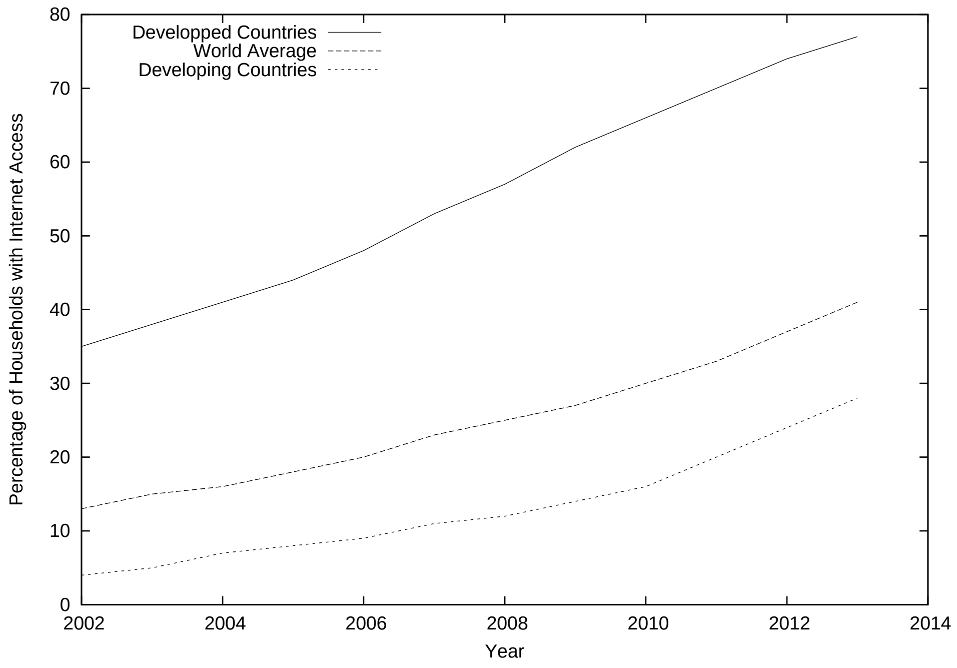


Figure 2.1: Increase in Internet access over the years according to statistics from International Telecommunication Union.

videos and send it to predefined points, instead of broadcasting.

2.2.2 Social Media

As Internet became widespread, users themselves have become a major source for data generation (see Figure 2.1). Nowadays, people share everything via social media: Twitter for what they think, Foursquare for where they are, Youtube for their videos, etc. With more than 2 billion Internet users, the data generated is huge. For example, Twitter announced in March of 2012 that they get 340 million tweets per day. Similarly, Facebook recorded 757 millions of daily active users in December, 2012, and Youtube gets 100 hours of video uploaded every minute.

As people use these data to communicate with each other, and share glimpses of their lives, that is not the only use of the data generated. In [2], researchers have shown that analysing tweets can result in incredible box office predictions, more accurate than prediction markets. Similar analyses are also being applied to stock market and election predictions.

2.2.3 Healthcare

In the process of medical diagnosis, doctors try to gather as much data about their patients as possible. They get blood tests done to extract physiological and biochemical statistics. They order electrocardiography to record electrical activities of the patient's heart. There are many other tests done on patients to gather states of many part of the human body, as every aspect of a living organism is a great data source. After that, doctors analyse the dataset to find out possible diseases and disorders. In its essence, this is gathering Big Data and analysing it to infer valuable information.

For example in [4], scientists developed a system called Artemis, that monitors premature babies' condition and provides clinical decision support for doctors. In the future, it is anticipated that such diagnostic and decision support systems will play vital roles in medicine.

2.2.4 Others

Every interaction with Internet is another data source for corporations. Consider the example of online shopping. At first glance, the data to be generated looks like user's activities such as browsing and buying. A basic dataset generated within an online shopping system would consist of entries that include the keywords users used for search, items browsed by users and purchases made. However, a lot of precious information lies within implicit correlations of these entries, such as how many items were browsed of the same kind before the purchase, demographic profiles of users that bought a particular item, etc. Mining the dataset wisely and making use of the relations of data, systems are built to make valuable inferences such as deciding if a user is only browsing, or actually intends to buy. This set of data is also used within recommendation systems that try to deduce it's users interests and suggests other items appropriately. We can observe the success of these recommendation systems on many websites, where we are offered clothes we might buy, movies we might like, people we might know, etc.

2.3 Storing Big Data

As we have discussed before, the number of data sources and data generation speed is enormous. This yields the problem of storing this data, and more importantly, accessing in a timely fashion. Since it is infeasible to develop supercomputers to handle both storage and access to data with discussed size, building cloud systems that consist of a number of computers running on commodity hardware connected via network is a common practice.

One of the main concerns when building these systems is *scalability*, which is the ability to add more hardware for extra storage. Distributed systems enable users to increase the number of computers (nodes) in the cloud system easily. Since the data is not stored in one location, but divided into chunks and distributed to different nodes, adding hardware does not affect existing data and requires almost no effort.

Availability is another point of concern, which is the ability to provide service even in case of partial hardware failures. To be able to continue serving even if some nodes malfunction, data is replicated and stored in more than one location. If any node fails, another node that keeps a copy of the same data takes over and serves requests. However, availability of all files are not guaranteed all the time, as the system might fail to serve a request in the unlikely event of malfunctions in all the nodes keeping a replica of the requested data. To increase the robustness of the system, the number of replications can be increased, providing higher availability at the cost of increased size. Hence, such systems takes the number of replicas as parameters to achieve required probability of availability specific to the application domain.

Latency is the time it takes the system to serve the requested data and can be crucial for real-time applications. Facebook's messaging system is a good example of a real-time system running on cloud storage [3]. Although there is time lost while reading / writing the data, the time lost while sending /receiving the data is the primary cause of latency, especially for geographically distributed systems. Hence, reducing the communication overhead is the main purpose while designing such systems. One can distribute replicas of the data such that the distance between the site storing the data and the source of request is close. In this case; however, writing speed will decrease, as the replicas will have to be sent to different and distant sites, again causing communication overhead.

By designing intelligent systems with swarms of computers that achieve what supercomputers would have difficulties achieving, *costs* are greatly reduced. Since these systems run on commodity hardware, Big Data applications are attracting businesses all over the world.

The most commonly used and known systems that offer distributed storage and processing capabilities are Apache's Hadoop Distributed File System (HDFS) and Google File System (GFS).

2.3.1 Apache's Hadoop Distributed File System

Hadoop Distributed File System is a file system built in Java language, and designed to run on very large clusters of commodity computers. Originally developed for Apache's web search engine project called "Nutch", HDFS is now a module in Apache's Hadoop project, which is defined as a software framework designed to allow distributed processing on large scale data-sets.

HDFS runs on a master-slave architecture that consists of a central controlling unit, called *NameNode*, and many *DataNodes* that store the actual data (see Figure 2.2). NameNode is a master server that controls file operations (such as create, delete, rename) and regulates access to files on the higher level. On the lower level, it manages replica placement and provides fault tolerance by monitoring states of DataNodes and redirecting any requests to appropriate nodes.

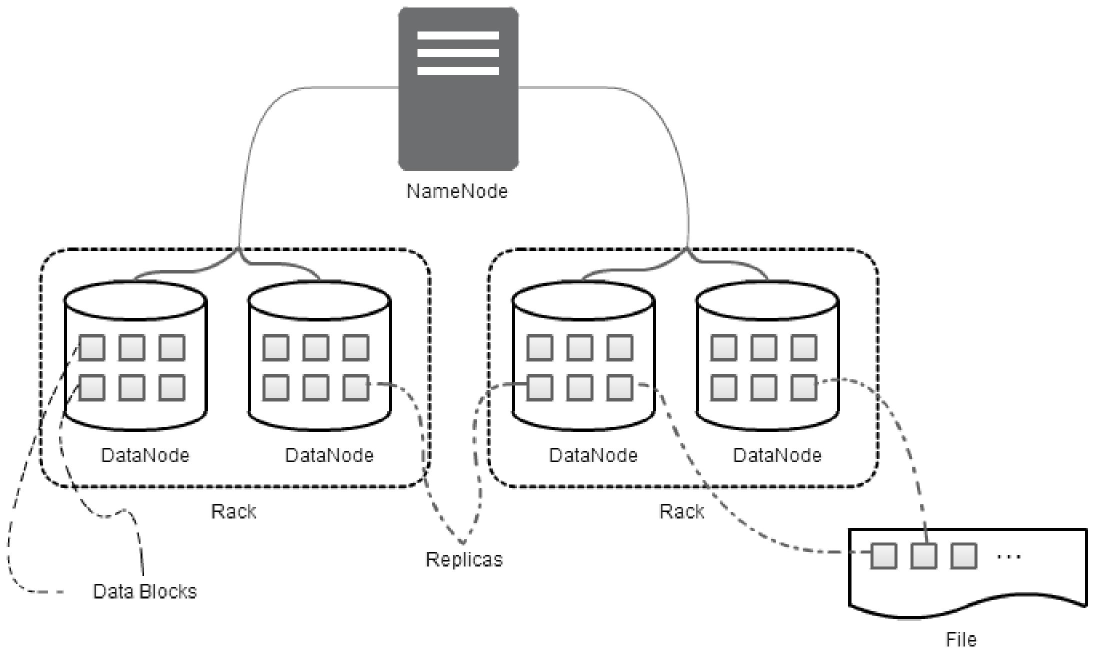


Figure 2.2: Hadoop file system architecture

DataNodes, on the other hand, are the computers that store the data and serve read / write requests. They also follow instructions received from the NameNode to create, delete and replicate data blocks.

Files on HDFS are split into blocks that are distributed on DataNodes. Each block is stored as a single file in the local system. When a file is created, NameNode partitions the data into a set of blocks, the size and the replication factor of which is configurable, and maps the write requests to DataNodes. As the DataNodes send heartbeat signals regularly, NameNode is always aware which DataNodes are available. In addition, thanks to Hadoop's Rack Awareness system, it also keeps track of which DataNode belongs to which rack, i.e. a branch in the network topology. With this knowledge, NameNode tries to place blocks of the same file into the same rack to reduce network traffic, but replicas into at least two different racks to increase availability, in case a DataNode branch is dead due to a network failure.

2.3.2 Google File System

Google File System (GFS) is a distributed file system developed by Google with main purpose of being used within Google. According to the article published by Google that covers the details and statistical analysis of the system [16], it was co-designed with applications and hence, it is designed to fit the requirements of Google's operations.

GFS runs on a master-slave architecture on clouds of Linux machines, just like HDFS. The design of the system is simplified by a single master server, referred as *GFS master* (NameNode in HDFS), that controls the overall mechanism. However, this master server tries to get involved with read and write operations as minimum as possible to avoid becoming a bottleneck for the system. Hence, no read or write operations are done through the master server. Instead, for example, when a client requires a read, it makes a query with the metadata to the master server, and master responds with the corresponding *chunk server* (DataNode in HDFS) information that stores the data to be read. The client caches this information and makes its read request to the chunk server directly, and continues to read data until the information about the chunkserver it cached expires. As all the data transportation occurs between the client and the chunk server that stores the data, master plays merely a controlling role with little involvement.

The expected size of files stored on GFS is more than 100 MB, often multi gigabytes. These files are divided into chunks (blocks in HDFS) and stored in chunk servers, where they are kept as regular Linux files. Each chunk is replicated through multiple chunk servers to maintain high availability in case of hardware failure, which is regarded as a common occasion instead of an exception. When a client makes a read request, master server returns the list of replica locations, instead of just one. If client discovers a failure that prevents communication with a chunk server, it continues on the list and tries the next replica. Clients are free to choose which replica to read from, and normally choose the closest one to themselves, which would reduce communication overhead.

When a client contacts the master server regarding a write request, the master server fetches the list of chunk servers that hold the chunk to write to. One of these servers is chosen as *primary*, and holds the lease for the chunk for a period of time, which can be extended as per primary's request. Upon receiving the list of chunk servers, the client pushes the data to be written to these servers. Once the data arrives and chunk servers acknowledge the event, the client sends a write request to primary. As there may be concurrent write requests, the primary serializes these requests by assigning consecutive serial numbers to each write request. Then, the primary redirects write requests to all replica holders, where the requests are applied in order of their serial numbers; hence, providing consistency among concurrent write requests.

Although the system works fundamentally the same way with HDFS, there are some key design differences that makes it meet the needs of Google's applications. For example, it is estimated that file changes occur mostly by appending new data to a file, instead of overwriting an existing one. For this case, GFS provides an operation called *record append* to append data to a given chunk atomically. This operation does not care about the offset at which the data is going to be written. Instead, it only ensures that the given data will be appended without its continuity being interrupted by any other concurrent write.

Another design choice is the size of chunks. It is chosen to be 64 MB, which is larger than usual file systems. Keeping chunk size bigger has advantages especially from Google's application perspective. It minimizes communication with master server as a file is distributed to less chunk servers, clients make less number of calls to master server to ask for chunk location. This is specifically efficient because most file reads and writes occur sequentially on large files. Larger chunk size also means less metadata to store at the master server, which affects master server's efficiency and enables the metadata to be stored in the main memory.

2.4 Processing Big Data - MapReduce

Storing Big Data efficiently is necessary but not enough as applications will naturally need computation and statistical analysis on data. As the data is so large that it is usually scattered onto clouds of computers, narrowing the computation down to one central unit leads to unacceptable efficiencies. Instead, distributing the computation process to computers to be done in parallel is the common way of approaching it. However, distribution of the computation is not a trivial task. In addition to design decisions such as where a computation should occur, there might be several constraints that arise from the characteristics of the computation.

For example, the computation may require running on a shared memory. In that case, the computation is split into different jobs that run in parallel on different parts of data. However, shared memory oblige lock mechanisms as jobs running concurrently on the same part of memory would result in race conditions. Therefore, such a job is required to lock a part of the memory before beginning of the computation, do the job, and unlock the memory when it's done.

On the other hand, most of the problems faced working on Big Data can be split into jobs that do the same type of computation on mutually exclusive sets of data. In that case, input data is sliced and fed to different jobs. Each job does the same computation in parallel and generate results accordingly. Then, these results are merged together to generate the output. As data in Big Data systems are usually scattered onto different computers, they are already sliced in a way. Since moving the data is more costly than moving the computation, jobs are run on the computers that hold the slices of data. However, not all problems can be split into jobs that can run in parallel. Some jobs may require outputs of other jobs. Such jobs would have to wait for other jobs to finish, receive their outputs and run after. If the jobs were running on a shared memory, it would be easier to program such conditions. But when the computation is running on different machines, problems of scheduling such preconditions makes it really complex to parallelize the computation.

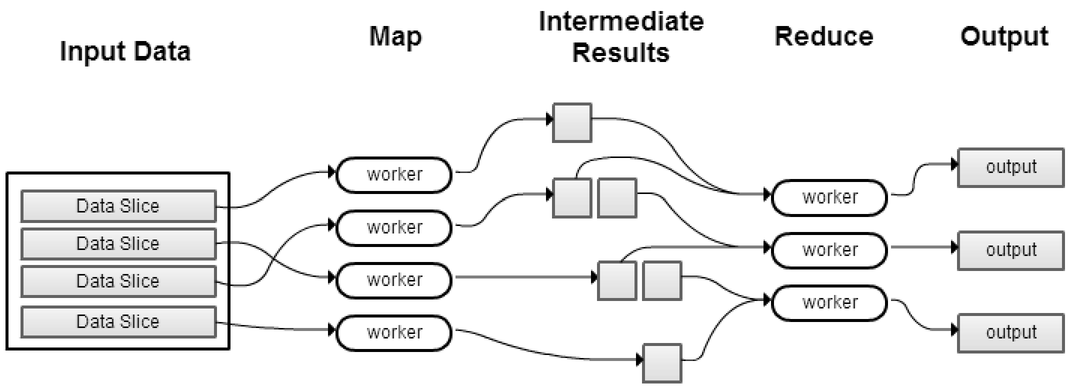


Figure 2.3: MapReduce structure

In 2004, Google introduced *MapReduce* [10] which is a programming model that simplifies the process of distributing computation and enables high-throughput calculations to be done on clouds of commodity hardware. The underlying architecture automatically handles low-level jobs such as distribution of input data, communication between workers, etc. Instead of dealing with these complex details, users are asked to implement two simple functions:

- $Map : M(k_i, v_i) \rightarrow P$, where P is a list of $< k_n, v_n >$ pairs
- $Reduce : R(k_j, V) \rightarrow Q$, where V and Q are lists of values

In *Map* function, input data is read as a key-value pair and an intermediate result is procuded as a list of key-value pairs. Then, these intermediate results are fed to *Reduce* function, where they are combined to create outputs. Different workers of Map might create intermediate results for the same key. When intermediate results are being sent to Reduce function, values of the results for the same keys are grouped into lists. By doing so, libraries ensure that all intermediate results of the same key are sent to the same worker that runs the Reduce function (see Figure 2.3)

Next, we give an example MapReduce application. Let us assume that we have a distributed database system that stores sales records for different sales people (see Figure 2.4). To calculate total amounts of sales by person, we introduce *Map* function given in algorithm 1.

Sales Person	Amount	Date
P_1	10	01.02.2014
P_2	20	12.02 2014
P_2	5	28.02 2014
P_1	15	05.03.2014

Sales Person	Amount	Date
P_1	5	05.02.2014
P_2	5	06.02 2014
P_1	15	05.03 2014
P_1	10	13.03.2014

Figure 2.4: Two fragments of sample collection of sales data

Let the number of workers running Map function be 2. Initially, a node is selected as master, that governs the process. Let us assume that master machine assigns map functions to workers $MW1$ and $MW2$ and specifies records on left and right as input for map functions to $MW1$ and $MW2$ respectively. Algorithm 1 will produce a list of $< P_i, A_k >$ pairs for every record, resulting in 4 pairs per worker, 8 pairs in total (see Figure 2.5a and 2.5b). These intermediate results are written to local files on map workers, splitted into R regions by passing keys to a hash function (see Figure 2.5c). After that, master notifies workers for reduce function, which are assigned to specific regions of the intermediate results and, in turn, reads their corresponding set of intermediate results. These results are then combined and passed to the reduce function specified by the user, which, for this example, would be the function given in Algorithm 2.

Algorithm 1: Map function for calculating total sale amount by person

```
1 Function Map(F : Collection Fragment)
2     foreach Record r in F do
3         emit(r.person, r.amount)
4     end
6     return;
```

Algorithm 2: Reduce function for calculating total sale amount by person

```
1 Function Reduce(<K, [V]>)
2     count = 0;
3     foreach value v in V do
4         count+ = v;
5     end
6     emit(count);
8     return;
```

This reduce function basically adds all sale amounts to find the total amount

Region	**Key**	**Value**
R_1	P_1	10
	P_1	15
R_2	P_2	20
	P_2	5

(a) Intermediate results of W_1

Region	**Key**	**Value**
R_1	P_1	10
	P_1	20
	P_1	15
R_2	P_2	5

(b) Intermediate results on W_2

Key	**Value**
P_1	{10, 15, 10, 20, 15}
P_2	{20, 5, 5}

(c) Input for *Reduce* function

Key	**Value**
P_1	70
P_2	30

(d) Overall output

Figure 2.5: Data at different phases of MapReduce computation

for a person. As the intermediate results are split into two for two salespeople, each reduce function sums up sales for only one person. As a result, two key value pairs, $< P_1, 70 >$ and $< P_2, 30 >$, will be the output of this MapReduce job (see Figure 2.5d). In the end, these 2 distinct outputs are written into two seperate output files. They are not automatically merged into one file, as usually results of a MapReduce computation are inputs for another one. If that is not the case, the receiver of these outputs is expected to handle fragmented output.

Although the given Map and Reduce functions generate the expected results just fine, there is still room for optimization. If we look at the intermediate results of map worker 1, we see two pairs being generated : $< P_1, 10 >$ and $< P_1, 15 >$ (R_1 in Figure 2.5a). This means that the map function emits two separate sale amounts for the same salesperson P_1. Instead of generating two intermediate results, we can sum up the amounts of the same person before sending them to reduce function, which is basically what the specified reduce function does. This way, the intermediate result would be $< P_1, 25 >$ and there will be only one pair, which, in turn, would decrease communication overhead. The function that does such partial reduce operation on map worker's intermediate results is called a *combiner function*. If the reduce function is both associative and commutative, the same procedure used in reduce phase can be applied to intermediate results as combiner function. However, there may be different ways of optimization by the combiner function based on the problem's nature.

2.5 Summary

Big Data is anticipated to be a big step for computer science. As data generation increases exponentially, data-driven decisions bring valuable implications on data into light. Successful applications of such data-oriented decision-making systems are spreading all around us: from systems that deduce box office success of a movie from the buzz in Twitter, to decision-support systems that help doctors custom-tailor elaborate diagnosis and treatment for patients.

Traditional systems fail to store and serve Big Data in applicable time limits. Usual Big Data systems require handling of huge amount of read/write operations per second, which is not affordable without doing parallel computations. Common approach is to use several commodity computers to divide and conquer these operations. Many similar applications, such as Google File System and Apache's Hadoop File System, offer a blackbox system that provides efficient read/write operations and provide high availability of data by handling hardware failures within themselves.

Although applications have specific complex patterns of computation and analysis, MapReduce paradigm, offered by Google, provides very useful abstraction and simplifies the job of programmers by handling data aggregation and communication among the data nodes itself, only requiring a couple of simple enough functions to be defined.

Chapter 3

Database Management Systems

3.1 Introduction

A database is simply a collection of organized data. Such a collection usually includes many different kinds of entities, number of samples from each kind of entity, and some relations among these entities. Any collection of data can be regarded as a database; such as, files stored by an operating system. For example, let us define directories and files as two entity types. Then, the most obvious relation among two entities, namely a directory and a file, is that a file is located inside a directory.

Database management systems are software applications designed to manage databases. They provide means of reading and writing data to clients such as users and other applications. In addition, some commercial applications handle maintenance jobs, which include backing up the data periodically, handling indexing, fragmentation, etc.

For decades, database management systems have been used successfully. However, with the explosion of Big Data, traditional systems began to fail at handling increased number of reads and writes. Engineers were forced to develop new technologies to provide new means of interaction with the data, and support distribution of the data.

A collection of data, a database, is organized based on the requirements of targeted operations. Some applications may be obliged to provide atomic write operations to ensure data integrity; whereas, others may require very fast access to read related data without the need to ensure data integrity. Such concerns are grouped under ACID term [17], which is an abbreviation of four properties: Atomicity, Consistency, Isolation and Durability. Atomicity means that a transaction either fails or succeeds as a whole such that it is not possible for some parts of a transaction succeed, whereas others fail. Consistency requires that transactions can not violate any rules or constraints defined by a database system. Isolation property ensures that concurrent transactions do not disrupt each other and result in a state as if those transactions occur serially. Durability means that when a transaction is committed, effects of it will hold, even in case of system failures. As a result, there emerged many different implementations of databases. They differ in terms of the ways they store data, and their read and write mechanisms. There are a number of different approaches available. We will go over the most common ones used.

3.2 Relational Databases

In relational databases, data is stored as a collection of tables (also known as relations). Each table stores a list of items of the same type, where rows in these tables correspond to items and columns correspond to different attributes of these items.

Relational databases are the most commonly used database models, according to DB-Engines, which is an initiative that collects and present DBMS usage statistics, supported by Solid IT company. Relational databases have been on the market for decades and used on many different types of applications. However, with the emergence of Big Data and social networks, their performance decrease in handling multi-level relations (such as friends of friends of friends) and difficulty

Id	Brand	Model	Year	Engine
1	Mare Motors	S22	2009	1.6L
2	Steed Cars	Shallow	2009	1.6L
3	Steed Cars	Quadus	2012	1.4L
4	Mare Motors	XQ15	2013	2.0L

(a) Table `models`

Id	Model Id	Model Name	Amount	Date
1	3	Quadus	15 000 $	20.05.2014
2	3	Quadus	14 500 $	26.05.2014
3	4	XQ15	22 000 $	13.06.2014

(b) Table `sales`

Figure 3.1: Sample database for a car sales company

they pose for scalability created a need for different models.

Relational databases implement relational model for the organization of data.

3.2.1 Relational Model

Relational model [6] defines the basic architecture and the organization of data for relational databases. In the context of relational model, a relation is not a connection among items, but a list of attributes that define an item. Looking at the example in Figure 3.1, there are two relations:

- $R_1(Id, Brand, Model, Year, Engine)$
- $R_2(Id, ModelId, ModelName, Amount, Date)$

All items that belong to a relation are required to follow the relation's schema strictly. There may be tuples that does not have a value for an attribute. In such cases, *null* becomes the value for the attribute of the tuple. Although lacking attributes is possible in this sense, there can not be tuples having more attributes than described in the schema. Hence, data stored in relational databases are said to be *structured data*.

Relational model requires a set of attributes to uniquely identify a tuple. This set of attributes is called candidate keys, one of which is selected by the user to be a *primary key*. This means that if values of attributes that are included in a candidate key is known, there is only one tuple having these values for the given attributes and it can easily be found. Note that there might be more than one set of attributes that defines a tuple. However, database systems require only one to be labelled as primary key. Every relation is obliged to have a primary key, which is unique for every tuple. This yields that there can not be two identical tuples, which is a necessity in relational model.

Although entities usually have a set of attributes that defines it naturally, they are not commonly used as primary keys in practice. Instead, surrogate primary keys are introduced which are not related to any data of the entity, but generated by the user of the database management system or by the system itself. The most obvious surrogate keys are natural numbers which are auto incremented with every insertion. Looking at the example given in Figure 3.1a, the set of attributes {Brand, Model} is enough to identify a tuple. (Note that we are assuming two brands can name their models with the same name. Just model would be sufficient otherwise.) However, we introduce a column named “Id” and assign unique integers to every tuple. This simplifies the database design.

It is certain that items on different tables will somehow be related. These relations are established by the use of *foreign keys*. A foreign key is set of attributes on a table that are primary keys on another table. Tuples in table given in Figure 3.1b have an attribute called *modelid*. This attribute is a foreign key for sales table, and a primary key for models table. To find out the engine volume of a car sold in sale with id 1, we look at the *modelid*, which is 3. Then, we locate the model with id 3 in models table, and find out that the car sold had an engine with volume 1.4 liters.

3.2.2 Normalization

Relations in relational model can be designed in many ways. This yields many possibilities of problematic designs. General tendency on database design is to avoid redundancy to prevent inconsistencies. Looking back at the car database example, each tuple in sales table includes both model name and model id. The pair <model name, model id> is redundant as it resides both in models and sales table. If the name of a model was to be changed, one would have to alter the tuple in models table, and all tuples in sales table that mentions the model. If a subset of tuples that mentions the model was to be missed, the database would contain inconsistencies. There will be <model name, model id> pairs in the database that shows different names for the same model id.

To avoid commonly made mistakes, normal forms have been introduced to act like a guideline for relational database designers [6][7][8]. Before getting into details of normal forms, let us define the terminology. *Functional dependency* is a relation between two sets of attributes, where a set of attribute practically decides the value of the other attribute. A set of attributes A is functionally dependant to another set of attributes B, denoted like $B \rightarrow A$, if each value combination for B yields exactly one value combination for A. For example, in a database where locations of university campuses are listed as $R(university, campus, town)$, for any combination of values for attributes $\{university, campus\}$, there exists only one town. Then, town is said to be functionally dependant on $\{university, campus\}$.

A relation is said to be in **first normal form**, if each attribute of the relation has an atomic value. For example, the table given in Figure 3.2a is not in first normal form, since attribute *color* of the tuple includes two values. To transform it to a relation in first normal form, the relation should be corrected as $R(team, color)$ and there needs to be two tuples: $\{A, red\}$ and $\{A, yellow\}$ (Figure 3.2b).

Second normal form is violated when a relation is not in first normal form, or an attribute that is not subset of the primary key is functionally dependant on a proper subset of the primary key. Primary key of the table given in Figure 3.3a is

Team	**Colors**
A	red, yellow

(a) Not in 1st normal form

Team	**Colors**
A	red
A	yellow

(b) In 1st normal form

Figure 3.2: First normal form

$\{university, department\}$. However, departments of universities have nothing to do with the rector of the university. Hence, field *rector* is functionally dependant only on *university*, making it a violation for second normal form. To fix this problem, we decompose the table so that field *rector* is in a new table with only the attribute it depends on: *university*.

University	**Department**	**Rector**
IEU	Computer Science	Prof. A
IEU	Fine Arts	Prof. A

(a) Not in second normal form

University	**Rector**
IEU	Prof.A

(b) In second normal form

University	**Department**
IEU	Computer Science
IEU	Fine Arts

(c) In second normal form

Figure 3.3: Second normal form

A relation is in **third normal form**(3NF) if and only if it is in second normal form, and all attributes of the relation are functionally dependant on only the key, nothing else. Figure 3.4a provides an example of a design that is not in 3NF. Primary key for this relation is *department* as the head of the department and his/her birthday can be determined by it. Although birthday of the head of the department is functionally dependant on *department* field, it is rather a transitive dependency, i.e. it actually depends on department head, and department head depends on the department. Hence, it is in clear violation of 3NF. To fix this violation, we decompose the relation into two, and make sure birthday is on a separate relation of which person (department head) is the primary key.

Boyce-Codd Normal Form (BCNF) is a normal form, also known as 3.5 Normal Form, that handles a special case at which 3NF fails. Basically, a relation is in

Department	**Dept. Head**	**Birthday**
Computer Science	Prof. B	05.11.1955

(a) Not in 3rd normal form

Department	**Dept. Head**
Computer Science	Prof. B

(b) In 3rd normal form

Person	**Birthday**
Prof. B	05.11.1955

(c) In 3rd normal form

Figure 3.4: Third normal form

BCNF if and only if all attributes on left hand side of non-trivial functional dependencies are candidate keys. We are not going into details as our exemplary dataset does not contain this case.

Normal forms discussed so far have been dealing with functional dependencies. **Fourth normal form** (4NF), on the other hand, focuses on the concept of multivalued dependency. There is a multivalued dependency between attributes A and B if each value of A yields multiple values for B, denoted like $A \rightarrow\rightarrow B$. In the example given in Figure 3.5, there is a multivalued dependency between attributes *person* and *phonenumber* ($person \rightarrow\rightarrow phonenumber$) as a person can own more than one phone. To store multiple phone numbers of a person, the address of that person mentioned more than once, which is a redundancy. To fix this issue, 4NF ensures that the relation is in 3NF and for each multivalued dependency $A \rightarrow\rightarrow B$ in a relation, A is a super key, i.e. it is a candidate key or any superset of it. Primary key in the given relation is $\{person, phonenumber\}$. Hence, multivalued dependency $person \rightarrow\rightarrow phonenumber$ violates the rule of 4NF. Normalization in this case is done by decomposing the table such that attributes *phonenumber* and the attribute it depends on, *person*, reside in another table, as in Figure 3.5c.

3.2.3 Reading and Writing Data

Data within a relational database is read and written by using Structured Query Language (SQL). Typical read query written in SQL has three parts : `SELECT`,

Person	**Address**	**Phone Number**
Sherlock	Baker Street 21B	555-15-25
Sherlock	Baker Street 21B	131-62-32

(a) Not in 4th normal form

Person	**Address**
Sherlock	Baker Street 21B

(b) In 4th normal form

Person	**Phone Number**
Sherlock	555-15-25
Sherlock	131-62-32

(c) In 4th normal form

Figure 3.5: Fourth normal form

`FROM`, and `WHERE`. `SELECT` statement corresponds to project operation (Π) in relational algebra. It defines the attributes that will be returned by the query. `FROM` clause defines the tables included in the query. `WHERE` clause states a propositional formula that filters the tuples to be selected, which corresponds to select operation in relational algebra.

For example, to get the names of models created in year 2009, we run the following query on the table given in Figure 3.1:

```sql
SELECT model
FROM models
WHERE year = 2009
```

The results will be two models $[S22, Shallow]$. The cost of such a read operation is basically the time spent on locating the data. Communication cost of sending found data back to the client is not a concern of query optimization, but rather the database management system.

The execution time of this query is highly dependant on the existence of an index on attribute *year*. If there is no index, database engine is required to look at all tuples within the table, locate the ones that fulfill `WHERE` statement, and project model attribute to return. The complexity of such a search will be in O(n). To avoid this full table scan, the concept of indexing is introduced.

Indexes are <key, location> pairs, which are stored sorted by their key. Hence, the complexity of an indexed search is in $O(\log n)$.

When the query requires to include more than one tables, **join operation** is used. For example, to find out the dates of, and the money made from sales of car model *Quadus*, the following query is required:

```
SELECT sales.date, sales.amount
FROM models, sales
WHERE models.name = "Quadus" AND
  sales.model_id = models.id
```

The result will be two pairs: $[20.05.2014, 15000\$]$ and $[26.05.2014, 14500\$]$. It is clear that two tables are scanned to produce this result. The process of scanning is optimized by an internal query optimizer of the database engine. There are three main join mechanisms: nested loop joins, merge joins and hash joins. When a query is submitted, query optimizer examines the query, and the tables included within the query. It generates a query execution plan based on its analysis on the query and tables, considering the number of rows in the tables, any indexes on the attributes used in join, selectivity of the given where clause, etc. Next, we look into the join operations in detail.

Nested loop join algorithm selects one table as inner table, and the other as outer table. First, outer table is filtered by any criteria that contains only the outer table. Then, for each row left on outer table, inner table is scanned for join condition. Rows from outer are matched with the rows of inner table and the union is returned. In worst case where no indexes exist, the join operation would require full table scan on outer table, and for each row in outer table, full table scan on inner table. Let m be the number of rows on outer table, and n be the number of rows on inner table; then, total complexity would be $O(m.n)$. However, indexes on attributes for both tables would increase efficiency a lot.

Merge join is a special type of join that requires both tables to be sorted by the attributes used in join. Algorithm scans through both collections in parallel and tries to avoid searching for tuples that do not possibly hold a value that can be matched. First, any unary eliminations are done. After that, algorithm starts scanning both tables row by row and keeps indexes to remember which row is scanned last for each table. At the beginning of the scan, each index points to minimum / maximum value in that table, based on the way the tables are sorted. Assume that there are two tables (T_a and T_b), and they are sorted on attribute F in ascending order. Initially, both indexes (i_a and i_b) would point to tuples that store the minimum values for F. Let v_a be the value for attribute F of the tuple that is being checked in T_a and v_b the value for field F of the tuple being checked in T_b. For any v_a, to find out which rows store matching values, algorithm checkes tuples in T_b as long as v_b is less than or equal to v_a. When i_b points to a row where v_b is greater than v_a, there is no need to continue scanning for the given v_a anymore, because tuples are sorted and the rows ahead would contain values for F, that are only greater than v_a. For any matching value, the two rows are matched and added to the result set. As tuples in both rows are only scanned once, complexity of this join would be $O(m+n)$, where m is the number of tuples in T_a and n is the number of tuples in T_b.

Similar to the merge sort algorithm, this little trick that prevents any unnecessary comparisons between tuples of two tables requires the tables to be sorted. For a table that stores millions of rows, merge join will be the most efficient, only if the table is stored in a sorted manner, or there are mechanisms to retrieve data sorted. Otherwise, the overhead caused by sorting millions of tuples would eliminate any efficiency merge join offers.

Hash join is the join type that is used when the tables are not sorted and cardinalities are fairly large. It consists of two phases : build and probe. Let F be the attribute on which the tables are joined. In build phase, one of the tables is selected as the build table. Then, the rows of this table are stored in a hash table, created by passing the values for F through a hash function H that is created automatically by the database engine, based on its statistical knowledge of the table. In probe phase, for each row of the second table, the value for F is passed through H to find the possible matches in the hash table. As there might occur collision in the hash table, rows that result in the same bucket are checked if they are matching. Let h be the complexity of hash function, then the overall complexity of the algorithm will be $O(h * m + h * n)$.

Main struggle of the hash join is that the hash table created in the build phase is required to be stored in memory. When the build table is too large, size of the hash table will exceed memory capabilities, resulting in use of virtual memory, and decreasing the efficiency of the algorithm hugely. Hence, it is common practice to select the table that holds fewer tuples as the build table.

There are two types of **write** operations: **update** and **insert**. An update operation is changing data on a tuple that already exists in a table; whereas, insert operation is adding a completely new tuple into a table. Two main factors of an update operation is the cost of locating the tuple(s) to be updated, which is basically the same cost as reading them, and the cost of changing some values of that tuple(s). On an insert operation, however, there is no searching for existing tuple. Hence, the cost only consists of the latter.

For example, to change the engine volume of model *S22* from 1.6L to 1.4L, the following query needs to be run:

```sql
UPDATE models
SET engine = "1.4L"
WHERE brand = "Mare Motors" AND model = "S22"
```

When executing such an update query, database engine first locates the tuples that matches the conditions in `WHERE` clause, then executes the operations in `SET` clause. As discussed in reading data section, the existence of an index on the attributes included in the `WHERE` clause of the query is the determinant factor of performance for the first part. However, when data is inserted or altered, indexes require maintenance, which results in extra work. As a consequence, having too many indexes on attributes included in `SET` operation yields low performance on update.

Insert operations, on the other hand, do not require any search on relations. As only new data is written into the database, cost of an insert operation consists of updating (or creating) required indexes, as previously discussed.

3.3 Document Based Databases

Document based databases emerged with NoSQL (not only SQL) movement. Although SQL databases have been used widely, their performance suffered for new types of queries. Having to follow a strict schema became too constraining.

The idea behind document oriented databases is to store semi-structured data in documents as a whole. A document includes all data regarding to an object, which corresponds to a tuple in RDMBSs. Data within a document is often stored as key - value couples. A set of documents stored together is called a collection, which corresponds to tables (relations) in RDBMSs.

Having semi-structured data relaxes schema constraints and lets data to be modelled more easily. However, not following a strict structure often results in maintenance problems.

Document based databases, (and other NoSQL databases) are usually not ACID compliant. Although there are some database systems that provide ACID properties, such as CouchDB, MongoDB is not one them. Instead, MongoDB ensures atomic operations on single documents and claim that it should be enough to ensure data integrity.

There are a number of formats used to store data in document based databases. Most commons are JSON (JavaScript Object Notation) and XML (Extensible Markup Language). As such formats have been widely used, there are a variety of libraries for all widely used programming languages that parses and encodes data in these formats. We will focus on JSON format and MongoDB.

```
{
  "name" : "Quadus",
  "year" : "2012",
  "engine" : "1.4L"
  "sales" :
  [{
    "amount" : "15.000$",
    "date" : "20.05.2014"
  },
  {
    "amount" : "14.500$",
    "date" : "26.05.2014"
  }]
}
```

Figure 3.6: Car model represented in JSON format

3.3.1 JSON

JSON is a highly flexible data format. There are four types of elements in JSON, which are objects, arrays, keys and primitive values. Objects in JSON are basically lists of key-value pairs. Keys in these key-value pairs are strings that acts just like hashes in hash tables. Values, on the other hand, can be other objects, arrays, or primitive values, such as number, string, boolean and null. Arrays in JSON are heterogeneous collections of objects, primitive values, or other arrays.

Objects are denoted with curly brackets ({ }), and arrays are denoted with square brackets ([ ]). A sample object can be seen in Figure 3.6.

This object has 4 keys, which are `name, year, engine, sales`. Fields `name` and `engine` are of primitive string type; whereas, `year` is stored as number. Field `sales`, on the other hand, stores an array of objects.

3.3.2 Document

A document in a document-oriented database is a collection of data that usually represents an object in real world. It is common practice to denormalize data and embed as much related information as possible into one document and store them together to avoid costly joins.

As data is denormalized on purpose for performance gains, normalization issues are ignored as a trade-off between consistency and operational efficiency. There is no universal rule of thumb when designing document oriented databases. It is preferred to have schemas, albeit relaxed ones and embedding subdocuments into documents is a common practice. However, the decision of embedding an object into another one, or having it as an individual document depends on the operations to be run on the data set.

When objects are represented in the form of a document, different types of semi-structured information can be embedded into one single document. Consider the representation of car model `Quadus` in JSON format in Figure 3.6. In addition to `name, year` and `engine` information, sales data is also inserted into the document. This yields the result that no join operations are needed when fetching sales data for a model. Instead, only model document is required to get both model details and sales information. Note that the same operation requires scan of two tables to join them in a relational database. While using this schema, it is possible to have indexed ids for models and have brand documents store it's models as an array of ids, for faster look-up for models through brands.

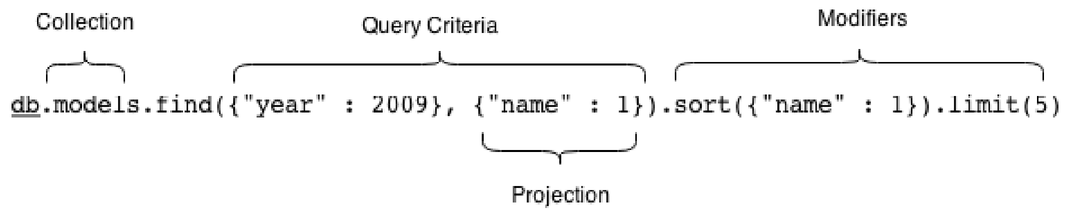
Another possible schema would be to have brands as master documents, and embed models into them as subdocuments. If the database is designed this way, to locate sales of a model, one would require either to know the brand that owns the model and locate the brand document to get the model and sales subdocuments, or search through brand documents and check if the models array includes the one being searched.

It is obvious that there are many number of design choices to be made and the decisions should be based on the search patterns of the application that will use this database.

3.3.3 Storage, Retrieval and Editing

Document based databases contain collections of documents. Documents are stored as a whole and documents in the same collection are stored together. Different document based database management systems provide different types of querying and editing interfaces. We will be focusing on MongoDB.

Queries in MongoDB consist of multiple parts: query criteria and modifiers. Query criteria is the part where a predicate is provided that chooses if a document will be included in the result set. Query criteria may also include a list of keys used for projection. Returned documents will include only the fields given in the projection list. Modifiers, on the other hand, are used to manipulate the result set, such as sorting the results, limiting the number of documents to be returned.



```
db.models.find({"year" : 2009}, {"name" : 1}).sort({"name" : 1}).limit(5)
```

Figure 3.7: Simple MongoDB query

For example, to get a list of models with year 2009, we need to run the query given in Figure 3.7. If only the names of these models will be shown, it is logical to project only the name part of the documents, as the query suggests. The results will be sorted on their names, and the result set will be limited to include 5 items max, as per modifiers given in the query. Note that value `1` given in projection and sorting modifier is used to state that the given field should be used in the operation and value `true` can be used with `1` interchangeably. If values `false`, or `0` are used instead, the fields mentioned will be excluded from the result. There are other modifiers, such as `skip`, that skips the given number of documents and returns the documents after that. Modifiers `skip` and `limit` are used together mostly for pagination.

Each query is restricted to search through the documents in only one collection. Hence, cross-collection queries, joins in relational databases, are not allowed, which can be limiting for applications working on MongoDB. To be able to query documents based on data from different collections, separate queries are required to run, the latter using the output of the former. For example, if sales data were to be stored in a different collection than the models, to be able to search sales data of all models created in a given year, we need to query first the models of the given year, and extract the model names from the first results, and use these models as input for the second query, which would match the model to sales documents and return the sales data.

Formerly mentioned relatively simple queries are not the sole ways of retrieving data. MongoDB provides three different aggregation operations that enables querying and manipulating data on server side. First one is fairly basic aggregation operations, referred as *single purpose aggregation operations*. This group includes simple functions that run on a single collection. For example, `count` is used to get the number of documents that returns true for a given predicate, or `distinct` is used to get distinct values the documents in the collection have for given key.

There are two other aggregation operations that provide more complex means of data retrieval. *Aggregation pipeline* is used when data being queried requires to be manipulated, grouped, processed and then returned. A good example for aggregation pipeline would be to sum the amount of money received for models, grouped by the year of the model that were made after 2005, for the given schema in Figure 3.6. To be able to get the data, we need to eliminate documents referring to models that were made before 2005, find all sales data grouped by years, and sum them up. For this operation, the query in Figure 3.8 will do the job.

Let us look closer to the query. The sales data of models that were built before 2005 do not concern us. Hence, we use `$match` operator to denote that the documents used in this aggregation need to have their year greater than or equal

```
    db.models.aggregate({
        $match : {"year" : {$gte : 2005}},
        $unwind : {"sales" : 1},
        $group : {
            "_id" : "$year",
            "totalSale" : {$sum : "$sales.amount"}
        },
        $sort : {"totalSale" : 1}
    })
```

Figure 3.8: MongoDB aggregation pipeline

to (`$gte`) 2005. Note that the `$match` operator will make use of the index on `year` field, if there is any. Then, we need to sum up all sales separately. However, each document contains an array of sales data, instead of one. Hence, we use `$unwind` operator. This operator duplicates documents to have only one sales data object on them, so that each sales data object is mentioned once and not in an array. As we need to group data by the model's year, we give `year` as `_id` for `$group` operation. MongodDB groups documents that generate same `_id` into one document, passing them through the other group operations, such as `$sum` operation given in the example. Finally, `$sort` operation sorts the generated results by the fields used as input, which yields sum of all amounts of sales, grouped by the year of the model, sorted in ascending order by the year of the model.

MongoDB also provides MapReduce framework. This framework makes use of the *MapReduce* paradigm discussed in section 2.4. It applies the map function to each document that matches the query criteria, emitting key-value pairs. Then, reduce function is applied to keys that have more than one value. As both map and reduce functions are custom javascript functions, it is easy to customize both procedures to generate complex results. It is also possible to state a finalize function to process and manipulate the results of the reduce function and shape it into a final result.

To perform the same operation given in Figure 3.8 in MapReduce paradigm, code presented in Figure 3.9 must be run on MongoDB instance. Note that the `mapReduce` interface given in MongoDB shell takes 3 inputs. First one is the mapping function that will be applied to all documents used in the MapReduce operation. Data contained in the currently processed document can be accessed using `this` keyword. As we are iterating on model documents, there is an array of sales for each model, and each of the amounts given in these sale elements must be included in the operation. Hence, we iterate on `this.sales` and for each sale data, we emit `<year, amount>` pair. It is obvious from this example that one document may emit more than one results for a map function. Then, these results are automatically grouped on keys, and given into the `reduce` function, which is given as the second parameter. This function iterates over the values, which are sale amounts for each sale, and sums them up to find the total amount received from sales of all models that were produced in the same year. The third parameter, which is an JSON object, is used to input optional parameters, such as finalize function, or a query criteria to select which documents will be included in this MapReduce operation. If no criteria is given, all documents will be passed through the map function. Other optional parameters include a “sort” field, which determines the sorting criteria of input parameters, or a “limit” which limits the number of documents that will be included in the operation, or an “out” value which indicates where the results will be written. It is possible to write the results of a MapReduce operation into a collection to be stored for later use, or to be given into a second MapReduce operation as input, thus enabling incremental MapReduce operations.

Aggregation pipeline is relatively more efficient than the MapReduce framework, as it has a coherent interface and provides predetermined operations; thus enabling internal optimization techniques run by the database engine prior to the execution. However, MapReduce framework accepts custom functions that provides a more flexible interface, with the additional advantage of being a widely-known concept.

Insertion in MongoDB is fairly simple, as objects are both represented and stored in JSON, and there are no schema restrictions on the database shell. To insert a document, function `insert` is used, which takes a single parameter. If

```
    db.models.mapReduce(
        //map function
        function()
        {
            var key = this.year;
            this.sales.forEach(function(sale){
                var value = sale.amount;
                emit(key, value);
            })
        },
        //reduce function
        function(key, values)
        {
            var result = {year : key, totalAmount : 0};
            values.forEach(function(value){
                result.totalAmount += value;
            })
            return result;
        },
        //optional parameters
        {
            query : { year : {$gte : 2005}}
        }
   )
```

Figure 3.9: MongoDB MapReduce

the parameter is a JSON object, one document is created in the specified collection, and the number of documents returned to inform the client for the result of the operation. If the parameter is an array of objects, bulk insertion operation occurs, returning an object called `BulkWriteResult` that contains detailed outcome including the number of documents created, any possible write errors.

Update operations, on the other hand, takes three parameters. First one is query criteria that is used, just like when retrieving objects, to find documents that will be affected by the given update. Second parameter contains instructions as to what the update operation will change. MongoDB provides obvious operators in addition to some very useful instructions to be used in an update operation. For example, `$set - $unset` operators are fairly simple and used to set/unset a field in the document. Operator `$addToSet`, on the other hand, inserts the given value into an array, only if the array does not contain the given value, thus easing the maintenance of sets. Options for the update operation is given as the third parameter. These options include "upsert" key, which lets the database system to insert a document with the properties given in the update instructions, if no document matches the query criteria. This option comes in handy as it lets the user avoid checking existence of a document before updating it, and inserting a new document when there is none. The other option is "multi" key, which implicates if the update operation might change multiple documents, or if the operation should only be run on the first document found.

3.4 Graph Databases

Graph databases are another part of the NoSQL movement that is becoming more and more popular. These databases model the data in a graph structure with vertices representing objects in the real world, and edges corresponding to the interactions between these objects. Nodes and vertices have properties, as key-value pairs, that are used to store details of objects and relationships.

Advocates of graph databases discuss the argument that when people model real world objects, they usually tend to use graphs. For example, entity - relationship (ER) diagrams are most commonly used diagrams in early modelling stages of software development. It is an easy way to abstract data and define how systems would work. As an ER diagram can easily be mapped into a graph structure, it is easier to model a graph database, rather than creating complex tables that require multiple joins, or choosing from varying possibilities of documents.

In addition to being a more natural way of designing a database system, graph databases are better in handling connected data. Consider a many-to-many relationship among objects of same type, for example, friendship among users of a system. To represent such a configuration, relational databases would require a new table which would include tuples for each connected objects, a table to hold unique ids of all <user, user> pairs. Finding objects that take part in such a relationship would require multiple costly joins of tables. If multiple depths of this relationship considered, such as friends of my friends, or friends that are friends with my friends, things would go out of hand. A document based database system might respond better to first depth objects in such a relationship, but again would fail with multiple depths. Graph databases easily handle this type of queries, as it enables queries to be graph traversals.

Moreover, graph databases make use of index-free adjacency, which means that relationships of an object directly point to other objects, instead of storing indexes for those objects. This avoids index-lookups and increases performance on multiple depth relational searches.

3.4.1 Graphs

A graph is a mathematical model that is used to represent objects and relationships among them. Objects in real world correspond to vertices (nodes) in a graph; whereas, edges connect these nodes and correspond to relationships among objects. Therefore, a graph is basically a set of vertices and edges, denoted as $G = (V, E)$. Edges might have directions, making the graph directed, or may be without directions, which yields in an undirected graph. Each vertex might be connected to any number of other vertices. However, edges are restricted to be connecting only two vertices. Therefore, for a relation that binds more than two objects, a new vertex is introduced to represent the action, and all other objects that take part in this action are connected via edges to this new vertex.

The number of edges that emanate from, or leads to a vertex v is said to be the degree of that vertex, denoted as deg(v).

A path between two vertices is a sequence of edges which starts from one vertex, and leads to the other vertex, without violating edge direction if the graph is directed. A cycle is a path where the two vertices the path connects are the same vertices, i.e., a path that returns to the point it starts. There may be more than one (infinitely many if a cycle is involved) paths between any two nodes. In an undirected graph, if there is at least one path from any vertex to any other vertex, the graph is said to be connected. In a directed graph; however, if any vertex is connected to any other vertex without violating direction constraints of edges, the graph is strongly connected. The connectivity of a graph implies the robustness of the graph and is calculated as the minimum number of vertices that must be removed from the graph to break the graph's connectedness.

When dataset of cars, models and sales, given in Figure 3.1, modelled as a graph, the result will be the graph given in Figure 3.10. Brands and models are obvious choices to be selected as vertices. First approach to sales, on the other hand, might be to have them as relations between customers and models. However, since there is no customer data available in the dataset for now, it is required to consider each sale an object, thus modelling them as vertices too. Note that the relations are relatively obvious in the given database, and the directions can be removed. However, usual tendency is to keep directions in data as much as possible, but ignore them in the context and treat the graph as undirected.

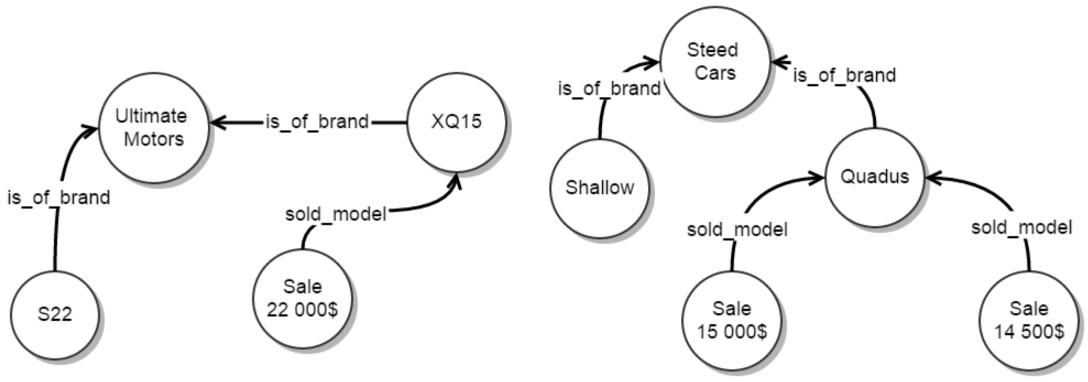


Figure 3.10: Car database as graph

There are many graph databases available, such as Titan, OrientDB, and Neo4j. We will be focusing on Neo4j, which is developed by Neo Technology Inc and used by thousands of companies, 50 of which is Fortune 2000 companies, according to their website [1].

3.4.2 Data Access

Various graph databases are now ready to be used in full-scale. However, when it comes to querying them, there is no consensus in the language used to access data. Unlike SQL being used in almost all of the relational database management systems, developers of graph databases decided to implement their own languages. As graph databases are relatively new technologies, there might be convergence to some of these languages; but currently it is not the case. As we will be focusing on Neo4j, we will look at Cypher language, which is used by Neo4j graph database.

As graph databases store data in vertices and edges, pattern in accessing the data is finding a set of vertices to begin with, and following a graph traversal query to get results. Consider the database schema given in Figure 3.10, to find the models of brand “Mare Motors”, first we need to find the vertex representing “Mare Motors”, and then find vertices that are connected with it with a ”is_of_brand” relation, resulting in “S22” and “XQ15” vertices.

Cypher queries usually consist of three parts: `MATCH`, `WHERE` and `RETURN`. `MATCH` clause is used to denote the graph pattern that will be searched throughout the database. This pattern includes nodes and relationships, using parenthesis “()” for nodes, and brackets “[]” for relationships. To get the models of a brand, like the example given earlier, one should run the following query.

```
    MATCH (n) <-[:IS_OF_BRAND]- (m)
    WHERE n.name = "Mare Motors"
    RETURN m.name
```

Figure 3.11: Basic Cypher query

In the cypher query given above, two nodes are mentioned in the `MATCH` clause, namely n and m. A relationship is also given between these nodes, which is labelled as IS_OF_BRAND. In the `WHERE` clause, we express a constraint, which says that node n should have name “Mare Motors”, which makes it match with the brand node of Mare Motors. Note that this constraint can also be denoted in JSON as `(n {name:"Mare Motors"})`, instead of using a `WHERE` clause.

Once this constraint is satisfied, all nodes that are connected to node n with an IS_OF_BRAND relationship will be models of Mare Motors. In the `RETURN` clause, the output of the query is given, which is name values of all vertices matching the pattern. It is possible to return nodes as a whole, by not stating a key in return clause. Note that although < denotes direction of the relationship, a pattern without a direction is also possible, which makes it match with direction of both sides.

Cypher is designed to resemble a graph as close as possible. Hence, query "(brand)<-[IS_OF_BRAND]-(model)" is highly readable, which makes it easier to understand existing queries, and improves retention rate of knowledge in this language.

As there are no collections, or tables, every object is mapped to a node in graph. For the formerly given example, the pattern tries to match all nodes with name "Mare Motors", even if they do not stand for brands. However, there is a natural grouping among vertices that represent same types of objects. To mimic this concept, vertices can be labelled in Neo4j. Same types of objects are labelled with same key, and these labels are used in patterns. This way, database management systems avoid full node scans and scan only the nodes with the label given in query. Similar to the former example, to find the number and total amount of sales of all models of brand "Steed Cars", the required query will be as the following, which will return <29 500$, 2>.

```
MATCH (n:brand) <-[:IS_OF_BRAND]- (:model) <-[:
   SOLD_MODEL]-(s:sale)
WHERE n.name = "Steed Cars"
RETURN SUM(s.amount), COUNT(s)
```

Figure 3.12: Cypher query with labels

Note that brand nodes are labelled as `brand` and model nodes are labelled as `model`. These labels are used in the query as (n:label), where n denotes the node, and label is the name of the label. Because there is no need for models in this query, node in the pattern for models is not given any name, just the label. In addition, as we are only interested in the number and total amount of money made in these sales, we use grouping operators `SUM` and `COUNT` to aggregate results.

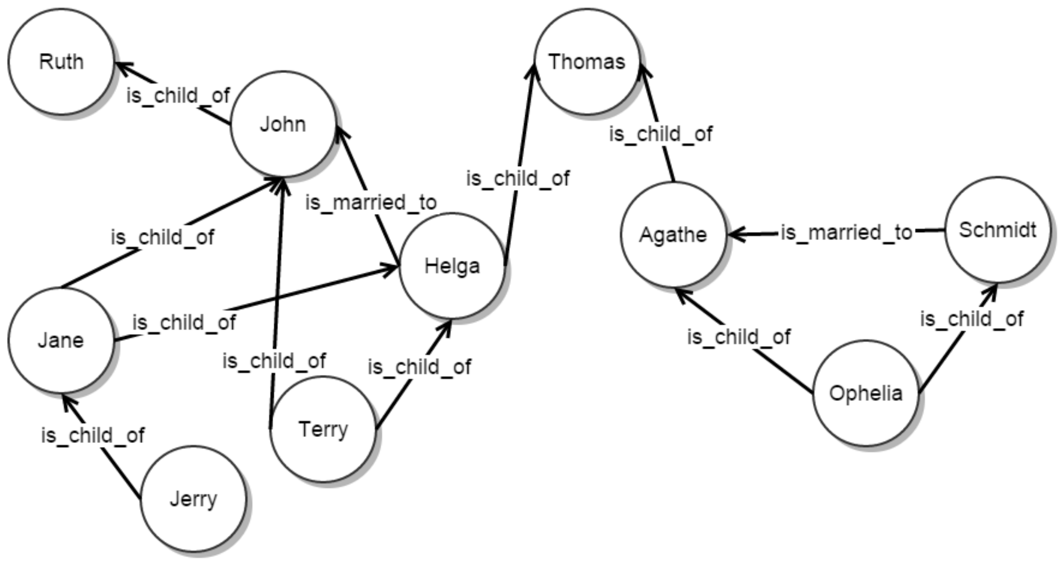


Figure 3.13: Family tree graph

Cypher provides more detailed means to query for more complex patterns. The power of graph databases comes forward while traversing paths of unidentified lengths. Consider the family tree graph given in 3.13, which represents a relatively small family tree with nine people, who are connected with marriages and parenthood relations. For example, to find all descendants of Ruth, we need to get all nodes that are connected to Ruth with relation `is_child_of`, regardless of the number of vertices in between. Following query does this job with making use of Neo4j's ability to handle variable path length selectors.

```
    MATCH (r{name:"Ruth"})
          <-[:is_child_of*]- (descendant)
    RETURN descendant
    LIMIT 25
    ORDER BY descendant.name
```

Figure 3.14: Cypher query with variable path length selector

Results will be Jane, Jerry, John and Terry. Note that the results are sorted by their names, as `ORDER BY` operator is used in the query. The other operator

```
    MATCH
       (ruth{name:"Ruth"}),
       (ophelia{name:"Ophelia"}),
       p = shortestPath((ruth)-[*..10]-(ophelia))
    RETURN p, LENGTH(p)
```

Figure 3.15: Shortest path query in Cypher

used `LIMIT` would limit the results to have only 25 nodes, if there were more than 25 vertices matching the pattern. Also note that the constraint for finding node for Ruth, that the name of the node is "Ruth" is given within the `MATCH` clause, instead of using a `WHERE` clause.

One of the most usable powers of Neo4j stems from its ability to search for a shortest path between two nodes. Cypher language provides the method `shortestPath()` that takes a pattern as a parameter, and finds the shortest path matching the parameter. For example, if we want to find the path that binds Ophelia to Ruth, we need to run the query given in Figure 3.15.

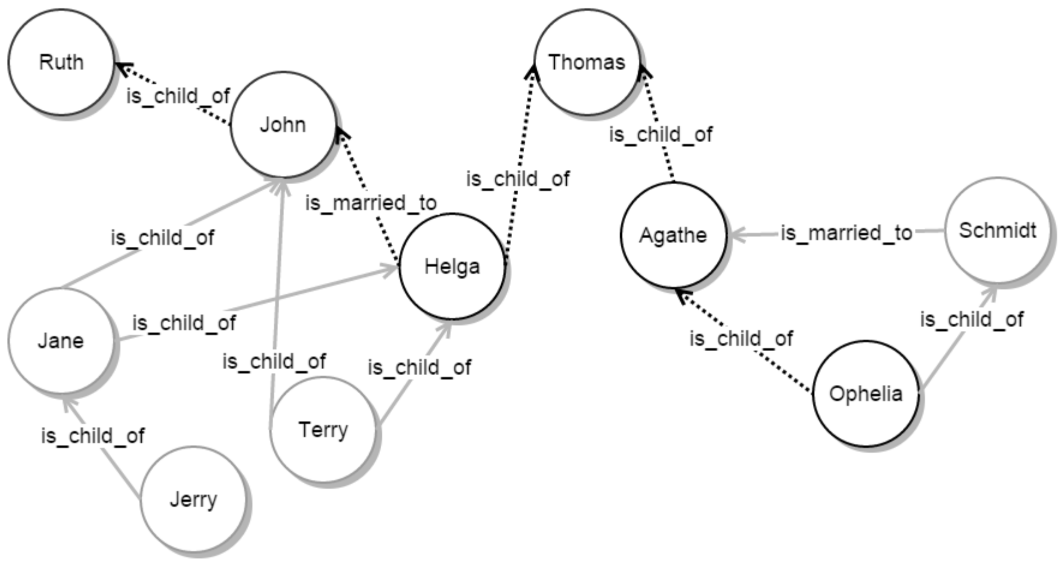
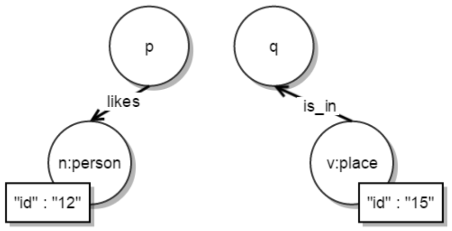
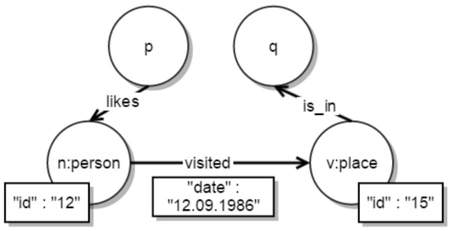


Figure 3.16: Shortest path between Ruth and Ophelia

Result set will include the shortest path between Ophelia and Ruth, denoted with the dashed lines in Figure 3.16, and the length of the path, which is 5. Note that vertices and edges not included in the path are greyed out for better



(a) Initial graph



(b) Graph with new relationship

Figure 3.17: Creating a relationship in Neo4j

visibility. Unique powers of graph databases become clearer when the difficulty of achieving same operation with other types of databases is considered.

3.4.3 Editing & Inserting Data

Data insertion in a graph database can be analysed in two sections: creating a vertex, or a relationship. Creating vertices is done by using the following query

```
    CREATE (n:new_node {"key" : "value"})
    RETURN n
```

In this query, `n` represents the node to be created. This node will have `new_node` as its label, and will be storing key-value pairs denoted within curly brackets. Optionally, this query returns the newly created node, as per instruction `RETURN n`.

To create a relationship, two vertices need to be located first. Then, the relationship is created between those two vertices. Consider the example given in Figure 3.17. Let's say that we want to create a `visited` relationship between vertices with ids 12 and 15, and we want this relationship to store "12.09.1986" as its `date` property. First, `MATCH` clause in the following query locates two nodes, denoted as n and v, based on the constraints on their ids (see Figure 3.17a). After that, `CREATE` clause creates a new relationship with the given type, `visited` in this case, with given key-value pair (see Figure 3.17b). Database management system returns these two nodes, and the newly created relationship, as per instructions given in `RETURN` clause.

```
    MATCH (n:person {"id" : "12"}),
          (v:place {"id" : "15"})
    CREATE (n)-[r:visited {"date" : "12.09.1986"}]->(v)
    RETURN n, v, r
```

To update already existing data within the graph database, cypher provides a simple interface. Similar to other databases, first the node or the relationship needs to be located. Then, using `SET`, or `REMOVE` keywords, data stored within these nodes and relationships are updated. The following query, for example, locates a node called `n1` and updates its data. `SET` keyword sets `newValue` as value of `newKey`. If the node already includes that key, value of it is changed. If no such keys exist within the node, new key is inserted. `REMOVE` keyword removes the key-value pair, key of which given as input.

```
    MATCH (n1:label {"id" : "id1"})
    SET n1.newKey = "newValue"
    REMOVE n1.oldKey
    RETURN n1
```

3.5 Summary

For years, relational databases have been used widely and hence, they hold a firmly defined structure. However, with the emergence of Internet, and social media, data generation and complexity increased and these relational databases began to fail serving efficiently. As they require strict schemas to be defined and followed, people felt the need to get rid of the schema constraints, since data generated nowadays are from various sources and do not strictly follow any rules.

With the emergence of NoSQL movement, various types of new approaches were developed to store data without heavy constraints of relational databases, with additional properties, such as horizontal scalability, being prone to sharding. Two intriguing technologies are document based databases and graph databases. Note that there are many variations of databases that were not mentioned in this section, as we focus on document based and graph based databases.

Document based databases tend to embed as much data as possible into one document to avoid costly joins of multiple tables. However, querying interface is thus limited not allowing any kind of join of more than one collection. Although they are more suitable for distributed storage and heavy computing among multiple servers using MapReduce framework, they struggle when handling connected data, and multiple degrees of relations. ACID compliance is also mostly lacking and hence, they might be insufficient to the area the database management system tries to serve, if reliability is a key issue.

Graph databases promote the idea that people tend to think in graphs when modelling. When data is stored as a graph too, there is no translation between the data model, and database schema, which in turn increase development efficiency. They are also very effective when objects are heavily connected and required queries include multiple depths of relations. In addition, Neo4j provides ACID transactions. But graph databases are relatively young and are under active development. Although they thrive at handling relationships, they perform weaker on simple data access.

In summary, each data model has its efficiencies and incompetences. They approach data differently and host characteristic designs. As a result, there is no one database management system that fits all project requirements. Instead, storage and access to data should be carefully planned and matching database model should be selected. We provide a detailed comparison among these database management systems in Section 5.4.

Chapter 4

Sınavo: A Big Data

4.1 Introduction

Sınavo is an online education system that provides its users means of studying in preparation for certain examinations. It currently supports users for LYS-YGS, which are examinations conducted by ÖSYM (Öğrenci Seçme ve Yerleştirme Merkezi), which is a governmental institution, to elect students for universities, or KPSS, which is another examination that is used in electing candidates of governmental employees. However, the concept is easily applicable to other areas.

The core mechanism in Sınavo is solving multiple-choice questions. Users can solve questions in many different conjunctures, such as in games, or in tests. However, social interactions play a significant part in Sınavo experience. Users can chat while playing games, share questions they struggle to solve to get opinions of other users, brag about their accomplishments in social networks such as Facebook, etc.

Sınavo has been in development for more than 4 years and is constantly adapting to new requirements of its users and the business. Initial version of Sınavo focused on solving questions and tests. In time, social interactions became available to users to provide a collaborative learning experience. After succeeding in attracting thousands of users, Sınavo started to provide course videos and video solutions to its questions. Now, Sınavo supports private schools to generate and evaluate tests, as well as tracking user statistics and analysing student performance.

4.2 Components

Sınavo provides different means of question solving. All questions in Sınavo database are linked to a collection of subjects which consists of subjects with 3 levels. First level is called lessons and is the most generalized category, such as Mathematics, Physics, Literature. Second and third level subjects are more specific. Questions are constrained to belong to three subjects, each from different levels. However, no question is allowed to be of more than one subject at the same level. In addition, subjects are strictly in a tree structure, meaning that each subject is a child of one and only one subject and there are no cyclic paths in the subjects structure.

The subject tree is an extensive collection of subjects with 9 lessons, and more than 170 subjects in second level, and more than 1150 subjects in the third level. The 3-level tree is a design choice, governed by the domain of the project. However, it can easily be changed to host other structures.

All questions are multiple-choice questions with exactly one right answer, four wrong choices. Users have three possible results when solving a question. They can solve the question correctly, wrongly, or skip it.

4.2.1 Question Solving

Most basic way of solving questions is a module called Question Solving. In this module, users select a set of subjects first and start a session. In a session, they are asked random questions from question database that belong to the subjects they have selected earlier. There is no sequential order in subjects of questions, and questions are primarily selected from the ones that the user have not solved yet. After user solves a question, he is shown statistics of that question, his results on that specific question, and his incremental statistics on the session. Statistics of a question include the number of times that question is solved, which is split into numbers of the question being solved correctly, wrongly, or left empty, and the fastest solvers of that specific question.

Sınavo has a score system that aims to encourage users to study. Top users with highest scores are listed on the main page. Users gain points by solving questions correctly, or might lose points if they solve questions wrongly. However, the outcome of the question solving action is not the sole component of points. The time that takes a user to solve question is inversely proportional to the score. Hence, the quicker the user solves a question, the more points he or she gets from that question.

4.2.2 Games

Games are the most popular parts of Sınavo system. They provide a competitive environment in which users are encouraged to solve more questions and be more careful when doing it. There are currently two types of games in Sınavo, which are BenBilirim and BildinBildin.

BenBilirim is a multi-player game that consists of rounds of question solving played in real time. In each round, all users are asked the same question and are required to solve it in a predetermined amount of time. When the time is up for a round, or all users have answered the question, the round ends and users are shown the result of that round. If a user solves a question correctly, he or she gains the right to continue to the next round. Users that did not answer the question correctly, which means that their answer was either wrong or empty or they did not answer the question in time at all, get an error. If the number of a user's errors exceed a predetermined limit, the user gets eliminated from the game, but is allowed to watch the proceeding rounds. The game continues until there is only one user left. If at any round all users fail to gain the right to go on to the next round, all users that contested in that round are allowed to proceed, as the game requires a clear winner.

Players are allowed to chat during the game, which provides a way of igniting social competition. They are also presented with a result page that shows the winner of the game with a positive visuals to provide a sense of achievement. The game requires at least two people to run, but there is no upper-limit to the number of contestants. More than 25000 games have been created in Sınavo. Most crowded games had more than 50 players and the longest games lasted more than 100 rounds.

BildinBildin, on the other hand, is a single-player game in which players try to solve as many questions correctly as possible. At the beginning, they are given a predetermined amount of time, and start solving questions. With each question they solve correctly, they gain extra time; whereas, solving a question wrongly results in a penalty in the time they have. The game goes on until they run out of time. Amount of time a player survives the game is that game's outcome. These records are visible on the records page, and present a good way of increasing competition among players, to motivate them to solve more questions.

4.2.3 Test

Formerly mentioned examinations (LYS-YGS and KPSS) have predetermined numbers of questions for each lesson. LYS and YGS happens once a year; whereas, KPSS can be taken twice in a year. Majority of students in Turkey attend to paid institutions that try to help in the quest of succeeding these examinations. These institutions prepare regular tests to both measure the students' current level, and prepare them for the real examination. As time is very limited in real examinations, students need to get faster at solving questions. These tests mimic the real examination in both time allowed to solve the tests and the number of questions included in a test.

These tests usually consist of a number of question groups, such as mathematics-1 or science. These question groups are predetermined sets of subjects (or lessons), some of which include only subjects of a specific lesson, whereas, some include more than one lesson. For example, the question group called *mathematics-2* consists of advanced subjects of lesson mathematics. On the other hand, question group called *science* includes questions from physics, chemistry and biology. Hence, there is no one-to-one mapping between the subject-tree and these question groups.

Sınavo provides a similar experience through tests. New questions are entered into the database to ensure that questions in a test are not solved before by students. Each test has a predefined time limit and predefined set of questions. System allows students to take these tests within a few days. For example, a test may be online for a week. When a students decides to take a test, he or she goes to available tests page, select the test, and begin solving. Once the students begins to take the test, there is no way of pausing the timer, so that each user has the same amount of time to finish. Once the time the test is online finishes, results of that test are announced. These results include statistics of the user on the test, such as the number of correct, wrong, empty answers, as well as their rankings in the set of all users that take the test.

Performances of students in such tests are highly valuable. Within this data, there resides strengths and weaknesses of students for each subject. As different users solve the same questions in the same amount of time, data generated is validly comparable. In addition, users take these tests in different time intervals, which makes the data decisive for tracking a student's performance change. If correctly mined, this data will outline performance track of the student, and may result in useful and accurate predictions.

4.2.4 Performance Analysis

Every action of users in Sınavo is stored to provide detailed statistics with informative presentations. For example, when a user solves a question, answer of the user is stored as well as the time it takes the user to answer, procedural statistics of the user's session, etc. Using this data, students can monitor their performances on specific subjects, compare it to average performances of all other users in selected subjects, see the fastest solvers of a question, and the time it took them to solve it, see how many times a question is solved, with details such as how many of that was correctly, etc.

Performance analysis section of Sınavo targets providing users the ability to monitor their current status and spot their weaknesses and strengths. Users can display numerical statistics of their actions in Sınavo such as the number of questions they have solved, or the number of questions they have yet to solve. In addition, they can decompose their statistics to see details on specific subjects on the subject tree.

In addition, users can display some fun statistics of their experiences in Sınavo. For example, they can see the number of people attended to a BenBilirim, which was the most crowded game the user attended to, or the longest lasting BenBilirim game and their rankings in that game. They can also check the person that beat themselves the most times, which makes that person a nemesis.

4.2.5 Landing Page

Landing page in Sınavo, which is the page users land after logging in, is a comprehensive dashboard. In the middle, posts of other users are displayed, which include updates from users, shared questions, or game results. This part is crucial for social interactions that occur within Sınavo. On the right hand-side, high scores and their owners are displayed. Displaying highest scores on everyone's landing page creates a crucial incentive for users to increase their scores and get a spot in the hall of fame. In addition, users are presented with their statistics of question solving in last 7 days. This way, users can easily see both their performances recently, and the amount of time they spent studying in Sınavo. Seeing a huge drop on last days is a good motivation for students to go on and do some work. They can also see their performances decomposed by lessons, with comparison to Sınavo averages. This provides a glimpse of their status on each lesson. As these statistics are displayed together with Sınavo averages, users can estimate their weak points easily, and focus studying on those.

All this data is generated in real time, meaning that every action that affects these statistics will be displayed on the page load. This is an important design choice, and a trade-off between performance and user-friendliness as seeing the displays change after doing some action is a good way of motivating users to keep up the work. Although caching some statistics and updating displays periodically would be more efficient from developers' perspective, when users do not spot any difference on these statistics after solving some questions, they tend to feel that what they have been doing was all in vain, and stop studying. This is especially the case when studying on a computer screen. People are used to study with pen and paper, and have fun on computer. Trying to study on a computer requires a good motivation and high focus. Hence, these tiny motivational designs choices play a huge role in keeping students' attention.

4.3 Challenges

Sınavo has been online for more than three years. Users have been using the system extensively, generating a huge amount of data.

As discussed earlier, Big Data is usually described with three V attributes : Volume, Velocity, and Variety. With millions of records stored in its database, Sınavo surely carries enough volume. New games are constantly created and played. In addition, daily averages of number of questions solved are close to six thousand. These facts prove that the velocity at which Sınavo system generates new data is high.

Variety can be considered as relatively weak in Sınavo, especially considering most of the actions can be seen as solving a multiple choice question. However, questions can be solved in many conjunctures, which means that each action yields different set of properties. This diversity is actually the main reason that makes Sınavo not very suitable for relational database management systems.

4.3.1 Handling Data

Sınavo has more than 80000 users signed up, more than half of which , around 47000, used their Facebook accounts to join Sınavo. These users can bond friendships on Sınavo, as well as importing their friendship data from Facebook. This generates a highly interconnected set of entities. Of course, not all of these users are active daily. Considering the fact that examinations occur annually, users that succeed in an examination are not likely to return to Sınavo. However, there are more than 30000 users that used Sınavo actively in the last month, which means that Sınavo endures a good traffic regularly.

There are more than 55000 questions in Sınavo database. These questions are stored as images to be printed on web pages, with one image storing the question part, and 5 others storing choices. To provide a better readability, especially for small devices such as mobile phones and tablets, these images require a certain amount of quality.

The biggest chunk of Sınavo data consists of data generated when users solve questions. This action has occurred for more than 6500000 times. This is an incredible amount of valuable data that holds vital statistics to be used, as well as a crucial source for data-driven decisions. A sample data of this collection includes a reference to user, to question and to the entity which encapsulates the session, which can be different games, or tests. It also contains the choice of user, the duration it took the user to solve the question, and the date at which this question solving action occurred.

In addition to data generated by actions occur in Sınavo, users also generate data via messaging tools and chatting in games. Sınavo users can send messages to other users to introduce themselves, get acquainted, and to converse. More than nine thousand messages have been sent using Sınavo. Users can also chat during competitive games. As these games are the most densely used features of Sınavo, chatting during these games is a major source for conversations. There are more than 300000 chat messages sent.

4.3.2 Accessing Data

As Sınavo is an online system working on request - response structure, response time is a crucial performance trait. Hence, accessing data in a timely manner is vital. This requirement dictates that no time-consuming data aggregations can be run on frequent requests.

Sınavo tries to summarize important statistics of its users on their landing page. These summaries include their statistics in last 7 days, or a status report on lessons. As studying in Sınavo is a continuous progress, users want to be able to monitor any changes in their statistics in real time. This enforces Sınavo to avoid caching these results and require that always up-to-date data is fetched when generating response. For example, when generating a status report on lessons, all the data required is present in the collection of solved questions. To be able to generate this report, user's success rate on questions of each lesson is required. However, fetching these from a collection of more than six millions of records is not an easy job. Even if there exist indexes, there might still be thousands of records need aggregation to generate the required results. In addition, different reports yield different criteria to filter solved question data; e.g., monthly success rate of users on subjects.

To be able to generate results in a nick of time, statistics are stored incrementally, instead of being calculated when requests arrive. This is done by storing a collection of statistics incrementally, and updating them when a question is solved. However, one collection of incremental statistics is not enough to cover all requirements. Queries to generate these reports include subject constraints to filter question solving statistics on only certain subjects, as well as time criteria to get monthly statistics of a user. To be able to serve statistics per subjects, records are generated per user per subject. This way, the look-up for one user's performance on a subject is locating a document with the specified subject.

Such reports do not consist only of individual user statistics. They also require a range of queries that generate statistics of all Sınavo users on questions of a subject, or, statistics of a question, regardless of the users solved that question.

To handle such queries, different collections store different incremental statistics. However, the number of collections to be updated when a question is solved is much like the number of indexes to be updated when an entity is introduced or updated in a database. When this number exceeds a certain amount, this design of incremental statistics begins working against the idea. Too many work to do when a simple entry is inserted into a collection will increase the insertion duration, which might yield in decrease in overall performance.

4.4 Summary

We have described Sınavo, which provides its users means of studying through a web interface. It includes different components, such as competitive games and tests to practice. The system encourages social interaction via sharing questions, chatting during games, etc. In addition, we have discussed numbers of Sınavo and shown that it endures a high volume data generation.

Chapter 5

Alternative Database Approaches to Sınavo

As discussed earlier, Sınavo has been in active development for more than 3 years, with some prototyping earlier than the start of this development. There have been two major design cycles that included different technologies running in the background.

Prototype of Sınavo was developed using ASP.NET web server with MSSQL database behind the curtain, which is a relational database management system developed by Microsoft Corporation. Not all of the systems current Sınavo has, was within the prototype's context. It was a showcase of what can be done for an online educational system, and how that can be implemented.

All versions of Sınavo was designed similarly to provide backward data compatibility. For example, users authenticate themselves using their emails and passwords. Passwords are stored in the database hashed with MD5 [24], which is a commonly used irreversible hashing function. Using an irreversible hashing function provides security over passwords because even if unauthorized parties get access to the database, they can not generate raw passwords from hashed values. When a user tries to authenticate, Sınavo system hashes the value provided by the user and checks if there is an entry in the database that matches given email-hashed password pair. As the data is backward compatible, in case of change in data storage, the only thing that is required to migrate the data is a mapping function that maps data in old schema with the new one.

The following sections discuss user-accessible versions of Sınavo built on different architectures with different database management systems. However, as mentioned earlier, Sınavo is a system that constantly evolves to fit the needs of both users and business. Therefore, not all versions include same features. For that reason, we will be discussing shared features and how they were handled, as our intention is to give a comparison among data models and their capabilities.

5.1 Relational Database

First version of Sınavo that was accessible to users was also built on an ASP.NET server, with MSSQL (Microsoft SQL Server) database. Hence, a relational model was used when designing data structure.

Web server interacts with the database via stored procedures which basically are functions stored in the database management system and called from the web server.

There were a few conventions followed when the relational database was designed for Sınavo. For example, each tuple has an Id field, which stores an auto-incremented integer and is used as a primary key. Although some of the relations have sets of fields that qualify for being primary keys, Id fields are introduced for ease of development. Moreover, naming of objects in the database are arranged so that similar items are displayed nearby. For example, all tables contain prefix "TBL_", and all stored procedures start with "SP_". In addition, tables storing similar data are named similarly. For example, `TBL_StatisticsPerQuestion` and `TBL_StatisticsPerSubject` are both tables storing statistical data. That's why names `TBL_QuestionStatistics` and `TBL_SubjectStatistics` are not chosen instead.

5.1.1 User data & Interactions

Sınavo stores many details about a user, on top of obvious properties such as names and surnames. For example, for authentication process, an email address and a password is used. To provide geographically constrained queries, addresses of users are also kept.

Users' authentication details and common properties of users of different types are hold in authentication table. Initial design included many user types, such as students, teachers, parents. However, they have not yet been used. To provide a single access point for all users, all authentication information needed to be held in a single table. The following relational schema is used for the table that holds authentication data.

`TBL_Auth(UserId, Email, Fullname, Password, Username, UserTypeId, RegistrationDate, IsAdmin, EmailValidated)`

In this schema, fields `Email`, `Password`, `Username`, `RegistrationDate` are self-explanatory. `Fullname` of a user is first name(s) and last name(s) concatenated. `IsAdmin` and `EmailValidated` fields are boolean fields stored as bit, former meaning whether the user is an admin, and the latter meaning if the user validated his/her email address, which is done by entering a code that is sent to the email address used in registration. `UserTypeId` is a foreign key, referencing `TBL_UserTypes` table, which has the following schema.

`TBL_UserType(Id, UserType)`

Table `TBL_UserType` serves enumeration purposes as field `Id` is an integer and `UserType` is a human-readable value, such as "Student", or "Teacher". It is possible to store `UserType` within `TBL_Auth` table in human-readable form. However, storing integers is cheaper than strings, and keeping human-readable strings in a single place is more reliable.

In addition, users can authenticate themselves via an email-password pair, or using Facebook's authentication API. Hence, a subset of users have their Facebook accounts linked to their Sınavo accounts. Although Facebook serves more information about users, Sınavo stores only their ids on facebook (`FacebookId`) and their emails used to login to Facebook (`FacebookEmail`). To store the extra data that comes with a Facebook account, the following table is used.

`TBL_FacebookAuth(`<u>`UserId`</u>`, FacebookId, FacebookEmail)`

Other details of users are kept in `TBL_UserInfo` table, and entities are matched by `UserId` fields in both tables. These details are stored in the following schema.

`TBL_UserInfo(`<u>`UserId`</u>`, TownId, MobilePhone, BirthDate, ProfileImage, GoalDepartmentId)`

In this table, profile images of users are kept as base64 strings, which is a format used for converting binary data into a string [20]. Here, `TownId` is a foreign key, denoting where the user lives, referencing table `TBL_Town`. Towns, or districts, are parts of cities; therefore, there is a many-to-one relationship between towns and cities. Hence, knowing the `id` of a town is enough to know the city in which the user resides. To comply with normalization rules, specifically third normal form, towns and cities are stored on individual tables with following structure.

`TBL_City(`<u>`Id`</u>`, Name)`

`TBL_Town(`<u>`Id`</u>`, Name, CityId)`

`GoalDepartmentId` is another foreign key in `TBL_UserInfo` and it corresponds to the university department user marked as his/her goal. Same normalization rule is applied here, and as `DepartmentId` is enough to identify the university, universities and departments are kept on seperate tables and only DepartmentId is mentioned in user info tuple. The following schemas are used for storing departments and universities.

`TBL_Department(Id, Name, SchoolId)`

`TBL_School(Id, Name, TownId)`

Note that `TBL_School` stores not only universities, but also other schools such as highschools. `TownId` in this table is used to locate a school.

It is obvious that due to relational model's schema constraints and normalization rules, information of a user have been split into many tables. This yields the problem of joining these tables to get all details of a student, which is constantly required when a web page is displayed, at least for the user displaying the page. The SQL query given in Figure 5.1 is used to generate these details of a user.

In addition, there are friendship relations created among users. To store this many-to-many relation, table `TBL_Friendship` is required. However, to form a

```sql
SELECT U.UserId as UserId, U.Fullname as Fullname,
   U.EMail as Email, C.CityName as City, T.Name as Town,
   I.MobilePhone as MobilePhone, I.BirthDate as BirthDate,
   U.Username as Username, U.UserTypeId as UserTypeId

FROM TBL_Auth U, TBL_City C, TBL_Town T, TBL_UserInfo I

WHERE U.UserId = I.UserId AND U.UserId = @UserId AND
   I.UserId = @UserId AND I.TownId = T.Id AND T.CityId = C.Id
```

Figure 5.1: SQL query used to retrieve the details of a user

friendship, first a user initiates a friend request to another user and they become friends only when the second user accepts this request. To store data about these actions, two tables with the following schema are required.

`TBL_Friendship(Id, UserId1, UserId2)`

`TBL_FriendshipRequest(Id, FromUserId, ToUserId, Accepted)`

When a user sends a friendship request, a new tuple is created in `TBL_FriendshipRequest` table, and the other user is notified of this request. When the second user accepts this request, `Accepted` attribute of the tuple created earlier is set to 1, and a new entry is placed into `TBL_Friendship` table that denotes the friendship of these two users. To find the list of friends of a user, that user's id is matched with `UserId1` or `UserId2` fields of `TBL_Friendship` table.

5.1.2 Questions - Subjects

As discussed earlier, subjects form a tree structure in Sınavo. Tree structures in relational databases are stored in a database with each item in the table as tuples, with a reference to their parents. Table `TBL_Subjects` complies with this structure with the following schema.

`TBL_Subjects(Id, Name, ParentId, LessonId)`

In this table, `LessonId` is a reference to `Id` field on the same table. Lessons, as discussed earlier, are subjects on the first level of the tree. This is a shortcut to find all subjects of a lesson, that would otherwise require two separate queries. Note that this is a violation of third normal form as `LessonId` is functionally dependant on both `Id` and `ParentId`.

`TBL_Question(QuestionId, Question, Answer, SubjectId, Date, IsChecked)`

Questions are stored in `TBL_Question` table, with schema given above, in the first version of Sınavo. As questions consist of a set of images and images are stored within database, these images were first converted to string in a base64 format, and all images and positional information is embedded into an xml file and stored in `Question` field. The correct choice is stored in `Answer` field, with possible values A, B, C, D, and E. `Date` is the date at which the question was entered into the database, and `IsChecked` is a flag to denote if the question is checked by administration of Sınavo to see if there were any problems with the question, such as bad image quality, or missing images. Questions and subjects are linked with `SubjectId` field.

Whenever a user solves a question, a new entry is made into table `TBL_QuestionSolved`, regardless of the context the question was solved in, such as games or tests. This entry includes id of the user that solved the question (`UserId`), the choice the user selected (`Answer`), the duration it took the user to answer to (`Duration`), and the date at which the user solved the question (`Date`). The context in which the user solved the question is enumerated and stored in `SolvedWhere` field. As a result, the following schema was used for this table.

`TBL_QuestionSolved(Id, UserId, QuestionId, Answer, Duration, Date, SolvedWhere)`

Using this schema, to find all the questions a user solved, one needs to get all entries in `TBL_QuestionSolved` which have `UserId` the same as that user's id. To check if a set of questions solved by the user was solved correctly, one need to query the question too, and check if `Answer` fields on both tables are the same or not. Note that dummy value 'X' is used if the user did not solve the question and left it empty.

5.1.3 Games

The first version of Sınavo includes both games, namely BenBilirim and BildinBildin. Data of a BenBilirim game is stored within the following table and schema.

`TBL_BenBilirim(Id, CreatorId, Name, SubjectIds, ChallengeDate, CreationDate)`

Obviously, `CreatorId` is the id of the user that created the game, and `Name` field stores the name of the game given by its creator. `ChallengeDate` is the date at which the game will start and `CreationDate` is self-explanatory. `SubjectIds`, on the other hand, are not marked as foreign keys, as the field stores all ids of the selected subjects on creation, as a single string with ids concatenated with comma as the separator. Note that this approach violates first normal form by storing more than one value in one attribute. Actual mapping of the game to subjects is saved into `TBL_BenBilirimSubjects` table with the following schema.

`TBL_BenBilirimSubjects(Id, BenBilirimId, SubjectId)`

This table is created to comply with the fourth normal form and handle multi-valued dependency. Filtering tuples with a certain `BenBilirimId` would give the list of subjects that were selected when game with that id was created. Questions asked in a BenBilirim game form the same mapping with multi-valued dependency, and are stored in a similar fashion, within table `TBL_BenBilirimAskedQuestions` with the following schema. Note that ordering these tuples by their ids would give the list of questions asked in a game chronologically; question that is asked in the first round being on top and question of the last round being at the bottom.

`TBL_BenBilirimAskedQuestions(Id, BenBilirimId, QuestionId)`

Attending a BenBilirim in the first version of Sınavo requires an invitation from either the creator of the game, or any other attendants. Users can either accept or reject an invitation, until the game starts. All these are inserted into the table that holds the members of a game called `TBL_BenBilirimMembers` with the following schema.

`TBL_BenBilirimMembers(Id, UserId, InviterId, IsAccepted, IsReminded)`

Here, `UserId` is the id of the user that is being invited to the game; whereas, `InviterId` is the person that invites the user. `IsAccepted` is a flag that stores whether this invitation is accepted, making the user a part of the game in turn, and `IsReminded` is a flag to store if the system had a chance to notify the user about this invitation.

`TBL_BenBilirimResult(Id, BenBilirimId, UserId, Round, Ranking)`

Results of a BenBilirim game are stored in table `TBL_BenBilirimResult` with the schema given above. Here, `BenBilirimId` is id of the game and `UserId` is the id of the user whose results are stored in the given tuple. `Round` field stores the round in which user got eliminated. Note that winner of the game would have the number of last round in this field, even if there is no actual elimination. `Ranking` stores at which rank the user finished the game, 1 being the winner of the game.

In the first version of Sınavo, BildinBildin games are not actual entities within the database. Instead, only their results are stored within `TBL_BildinBildinResult` with the following schema.

`TBL_BildinBildinResult(Id, UserId, DurationSeconds, CorrectCount, WrongCount, BeforeRanking, AfterRanking, Points)`

Here, `UserId` is the id of user that played the game, `DurationSeconds` is the time the game lasted in seconds. `CorrectCount` and `WrongCount` are the numbers of correct and wrong answers of the user during the game, respectively. Points is the points user got from the game. `BeforeRanking` and `AfterRanking` are the rankings of the user before and after the game. As Sınavo stored only the highest record of any user for `BildinBildin`, these fields are used to generate more detailed result report, and to show the user the affect of playing that specific game in his position in records. These records are stored in `TBL_BildinBildinRecords` table with the following schema, where `RecordSeconds` is the highest duration of BildinBildin game the user with id, stored in `UserId`, played.

`TBL_BildinBildinRecords(UserId, RecordSeconds)`

5.1.4 Tests

Tests in the first version of Sınavo are stored within table `TBL_Test` with the following schema.

`TBL_Test(Id, BeginDate, EndDate, Duration, Name, Description, SolvedCount, Type)`

`BeginDate` and `EndDate` fields store the date interval in which the test can be taken by the users. `Duration`, on the other hand, is the number of minutes users have to solve the test, once they start. `Name` and `Description` are self-explanatory. `SolvedCount` is an incremental shortcut to store the number of times a test is solved. `Type` is an enumeration for the different types discussed in section 4.2.3. There exists a many-to-one relationship between questions and tests. This relationship is stored in `TBL_TestQuestion` table with the following schema.

`TBL_TestQuestion(Id, TestId, QuestionId, SubjectId)`

This table also contains a violation of third normal form, as `SubjectId` field depends on the question, id of which is stored in `QuestionId`. However, not storing id of the subject here requires a lot more joins to get subject of each question.

`TBL_TestUser(Id, TestId, UserId, QuestionId, SubjectId, Answer, Duration)`

Actions of users in a test are stored in table `TBL_TestQuestion` with the schema given above. This table also contains the same violation of third normal form. Again, it is designed so to reduce number of joins required. `Answer` is the choice user chooses, and `Duration` is the number of seconds that takes the user to solve that specific question. For each question in a test, there will be a tuple entered into this database once a user solves that test. However, storing data only on this table would require huge amount of work when a result for the given test needs to be calculated. To avoid this work, as once the user is done solving the test there will not be any changes regarding that user's data, a summary is calculated and inserted into the table `TBL_TestUserStatistics` and `TBL_TestUserStatisticsPerSubject` with the following schema.

`TBL_TestUserStatistics(Id, UserId, TestId, Total, Correct, Empty, Wrong, Net, Ranking)`

`TBL_TestUserStatisticsPerSubject(Id, UserId, TestId, SubjectId, Total, Correct, Empty, Wrong, Net, Ranking)`

These tables contain summary of a user's performance on a test, which is sufficient for display purposes of Sınavo 1.0. The only difference between two tables is that `_TestUserStatisticsPerSubject` stores data divided into tuples that correspond to user's performance on only one subject; whereas, `_TestUserStatistics` stores overall statistics generated by the user on that test. Net field on these tables store a net value for the user. It is a common evaluation used in tests done in Turkey which is calculated subtracting 0.25 from the total number of correct answers of a user, for each wrong answer. Hence, on a test where statistics shows, for example, 10 correct answers and 3 wrong answers, user's net value would be 9.25. This is again a redundant data, only used to avoid this calculation.

5.1.5 Statistics

As mentioned earlier, all data regarding users' performances are kept within a single collection; namely, `TBL_QuestionSolved`. As a result, all statistics regarding users' question solving performance can be extracted from this table. However, as there are thousands of questions solved in Sınavo daily, and statistics get displayed more times than that, recalculating everything on the run is not a viable option. To avoid this heavy calculation, statistics are incrementally kept in the database and when the need arises, getting these statistics is a simple fetching from a table.

`TBL_StatisticsSubjectPerUser(Id, UserId, SubjectId, Total, Correct, Empty, Wrong, Net, Duration, Month, Year)`

Table `TBL_StatisticsSubjectPerUser`, with schema given above, stores monthly statistics of a user (referenced by `UserId`) on a specific subject (referenced by `SubjectId`). It stores month and year on separate fields to ease the process of aggregating tuples to find a user's statistics for a year. `Total` field stores the total number of questions solved by that user on that subject in the given month. Number of questions solved by the user correctly corresponds to `Correct`, wrongly to `Wrong`, the number of questions left blank are stored in

`Empty` and the net value is stored in `Net`. `Duration` holds the total number of seconds a user spent solving questions on that subject.

User statistics are not the only incremental statistics stored. To store statistics of all users on a specific question, table `TBL_StatisticsPerQuestion` is used with the following schema.

`TBL_StatisticsPerQuestion(Id, QuestionId, Total, Correct, Empty, Wrong, Net, Duration, Month, Year)`

Similar to incremental statistics mentioned earlier, `Month` and `Year` refers to the date at which these statistics occurred. `Total` is the field that stores the number of times that specific question is solved within the given year and month. `Correct`, `Wrong`, `Empty` are the number of times users solved that question correctly, wrongly, or left it blank, respectively. `Duration` is the total number of seconds spent on this question. Similarly, there is a table called `TBL_StatisticsPerSubject` that stores monthly statistics of questions solved by all users on a specific subject, with the following schema.

`TBL_StatisticsPerQuestion(Id, SubjectId, Total, Correct, Empty, Wrong, Net, Duration, Month, Year)`

5.2 Document Database

Second version of Sınavo uses Node.js platform, which is a javascript platform built on Google Chrome's javascript runtime with a non-blocking event-driven IO system. This web server stores data on a MongoDB database, as documents.

MongoDB automatically creates ids for each document inserted into database. All documents have "_id" fields that are required to be unique, and act as a primary key. Developers can present different types of data for this field, such as String or Number. MongoDB uses the type ObjectId, if non specified. This is a special structure for MongoDB that stores the time the document is created and some other parts that makes it unique within the database.

5.2.1 User data & Interactions

MongoDB supports semi-structured documents, which comes in handy while developing a system to support entities that share some parts but differ in others. Information of users is a good example for this kind of variance, as users might have facebook accounts and need data about their accounts on facebook stored too. As a result, when defining documents, all possible fields are presented here, leaving the responsibility of checking if such fields actually exist on a document to the developers.

User documents are structured with the following schema.

```
users : {
  _id : ObjectId,
  emailValidated : Boolean,
  username : String,
  fullname :String,
  email : String,
  password : String,
  facebookId: Number,
  facebookEmail: String,
  isAdmin : Boolean,
  lastLogin : Date,
  dateJoined : Date,
  city: { type: ObjectId, ref: "City" },
  district: { type: ObjectId, ref: "District"},
  birthdate : Date,
  telephone : String,
  school : {type:ObjectId, ref:"HighSchool"},
  target  :   {
    university : {type: ObjectId , ref: "University"},
    department : {type: ObjectId , ref: "Department"}
  },
  isBanned : Boolean,
  friendshipRequest : [ {type: ObjectId, ref: "User"}],
  friendshipWait: [ {type: ObjectId, ref: "User"} ],
  friendship : [ {type: ObjectId, ref: "User"} ],
}
```

As it can be seen from the schema, data about a user is collected into a single document as much as possible. It includes a wide range of data. For example, in the array of `friendshipRequest`, all ids of users that asked to be friends with this user is stored. Field `friendshipWaitFriendship`; on the other hand, stores the users to whom this user sent a friendship request. So, when a user sends a friendship request to another user, id of the other user is inserted into the documents of both requester, and requestee. The array friendship holds references to people with whom the user is friends with. In target, references both to the university and the department that the user targets are kept.

As a result of such comprehensive document design, getting a user document generates enough data for most of the user-oriented operations. Because queries can include projection; i.e., fetching only required parts of documents, storing all this data within a single document does not cause inefficiency. In addition, all data is stored in one collection does not require any joins to fetch them. This yields in ease of use, especially when compared to relational schema in which these operations required more than 5 different tables and joins.

As mentioned earlier, some users of Sınavo use their facebook accounts to authenticate themselves; whereas, others use their email - password pairs to login. By the ability to support semi-structured data, users collection can store both users with `facebookId` and `facebookEmail` and the ones without them.

The city and district the user resides in are stored in collections called cities and districts which store documents of following simple schemas.

```
districts : {
  _id: ObjectId,
  city_id: ObjectId,
  name: String
}
```

```
cities : {
  _id: ObjectId,
  name: String
}
```

5.2.2 Questions - Subjects

The way the subjects and subject tree is stored in MongoDB is much like in relational database. Subject documents correspond to individual subjects, with a reference to their parents, stored in parentId. A subject document is structured with the following schema.

```
subjects : {
  _id: Number,
  parentId: Number,
  name: String,
  numberOfQuestions : Number
}
```

The reason that `_id` and `parentId` fields are of type number, instead of ObjectId, is that subjects have been imported from the relational database, where ids are auto-incremented integers.

On the other hand, the way questions are stored is changed drastically. One major change is that instead of storing base64 images within the database, storing images is delegated to Amazon's S3 (Simple Storage Service) which is an cloud-based object storage service. Image parts of questions are stored within different buckets in Amazon S3, named after question's id. As a result, when a question is displayed, only urls to the images are sent to the client, after which, client contacts with Amazon S3 servers to download them. That is why only the image coordinates are stored in the database, regarding the visualization of the question. Questions are stored in the database with the following schema.

```
questions :{
  _id : type : Number,
  subjects : [{ type: Number , ref:"Subject"}],
  answer : String,
  qx : Number, qy : Number,
  ax : Number, ay : Number,
  bx : Number, by : Number,
  cx : Number, cy : Number,
  dx : Number, dy : Number,
  ex : Number, ey : Number,
  total : Number,
  correct : Number,
  wrong : Number,
  empty : Number,
  duration : Number,
  isChecked : Boolean,
  isFaulty : Boolean,
  difficulty : [Number],
}
```

One-to-many mapping form questions to subjects are stored within the subjects field of a question document, which is an array of subject ids, with three items in it. Fields `qx-qy`, `ax-ay`, `bx-by`, `cx-cy`, `dx-dy`, and `ex-ey`, are x-y coordinate pairs of images of questions and all 5 choices, respectively. As it is common practice to embed as much data as possible into the question document regarding that question, incremental statistics of the question is also stored within its document. Field `total` is the number of times the question is solved; where `correct` is the number of correct answers and `wrong` is the number of wrong answers given by the students. `Empty` is the number of times students left the question blank, and `duration` is the total number of seconds spent by users on the question. Similar to relational schema mentioned earlier, `isChecked` is a flag storing if the question is checked by Sınavo administration for visual problems, and `isFaulty` is a flag storing if the question is faulty.

`Difficulty` is an array of enumerations calculated by the system, as questions do not strictly belong to one difficulty class, but hold an interval which might include multiple difficulty levels.

Similar to relational model, whenever a question is solved by a user, a document is inserted into a collection called `solvedQuestions` with the schema given below.

```
solvedQuestions : {
  _id : ObjectId
  date : Date,
  user : { type: ObjectId, ref: "User"},
  question : { type: Number, ref:"Question" },
  subjects : [ {type: Number, ref:"Subject"} ],
  answer :
    {type: String, enum: ["A","B","C","D","E","X"]},
  duration : Number,
  result :
    {type: String, enum: ["correct", "wrong", "empty"],
  where :
    {type: String, enum: ["sc", "bb", "bildin","d"]},
  test : { type: ObjectId, ref: "Test"},
  testQuestionNo : Number,
  challenge : { type: ObjectId, ref: "BenBilirim"},
  challengeRound : Number,
}
```

A solved question document stores references to the user, in field `user`, and to the question, in field `question`. Subjects of the question is also stored within an array, called `subjects`, to support subject oriented aggregations easily, without requiring to populate questions. The choice user chose is stored in `answer` field and the seconds it took the user to choose it in `duration` field. Field `result` stores the result of this question solving action, which can be correct, wrong or empty. As this collection stores any question solving action, regardless of its context, where the user solved the question is stored in `where` field; values of which correspond to in question solving interface (sc), benbilirim game (bb), bildinbildin game (bildin) and a test (d). Again, as the schema is semi structured, only the documents corresponding to questions solved in a test would have fields `test` and `testQuestionNo`; and only the documents corresponding to questions solved in benbilirim would store challenge id in `challenge` field and the round in which the question was solved in `challengeRound` field.

There are some redundant data in this collection, such as the subjects of a question, or the result of the action, which could be calculated by comparing field answer with the correct answer stored in question document. This is because when generating complex statistics, an aggregation operation is run on this collection. And as aggregation operations do not allow usage of multiple collections, embedding redundant data to this document becomes necessary. As mentioned earlier, the school of thought behind NoSQL supports data redundancy compared to having strict schemas and constraints, to gain performance.

5.2.3 Games

The structure behind games is changed when Sınavo moved on to the second version. Instead of relying on a third party application, such as Adobe Flash, which is the way Sınavo 1.0 handled things, to maintain a constant connection between the users and the system, Sınavo 2.0 uses web sockets to push events to clients and support request - response structure of web perfectly. This change affected the way data is stored within the database.

Ben bilirim games are stored in `benbilirims` collection with the following schema.

```
benbilirims : {
  _id : ObjectId
  name : String,
  subjects : [ {type: Number , ref:"Subject"} ],
  createdDate : Date,
  creator : { type: ObjectId, ref: "User"},
  maxErrors : Number,
  state :
    { type: String , enum: ["NotStarted","RoundBreak",
       "Running", "Finished"], default: "NotStarted" },
  currentRound : Number,
  currentRoundQuestion :
    {type: Number, ref: "Question"},
  participants : [ {type: ObjectId, ref: "User"}],
  stats : Mixed,
  solvedQuestions :
    [ {type: Number, ref : "Question"} ],
  startDate : type: Date,
}
```

Again, all data regarding a game is stored within a single document.

`Subjects`, for example, is an array of subject ids that store the list of subjects selected when creating the game. In addition, as the game proceeds, field `state` is changed to reflect the current state of the game. Field `solvedQuestions` stores an array of question ids that are asked during the game and `startDate` is the date at which the game is planned to start.

The most interesting part of a benbilirim document is the field labelled `stats`, which is of type Mixed. Mixed values are JSON objects that do not follow any schema. For example, if a game is played with only 2 people, `stats` field would store only the winner and the second. Otherwise, it would contain all top three players, and other participants that are eliminated in earlier rounds and did not make it to first three positions. As the second version of Sınavo includes a separate application that acts as a server for BenBilirim games, individual rounds are not stored separately. Instead, only the results of a game is stored within this `stats` field.

Similar to BenBilirim games, BildinBildin games are changed too, to support request - response structure with discontinuous connections. A BildinBildin document is structured with the following schema.

```
bildinbildins : {
  user : {type: ObjectId, ref: "User"},
  subjects : [ { type: Number, ref:"Subject"} ],
  createdDate : Date,
  solvedQuestions :
    [ {type : Number, ref : "Question"} ],
  remainingDuration : Number,
  totalDuration : Number,
  correct : Number,
  wrong : Number,
}
```

Whenever a BildinBildin game is played by a user, a new document is created in `bildinbildins` collection. A BildinBildin document stores the subjects user chooses when creating the game in `subjects` field, and the questions solved during the game in `solvedQuestions` field as an array of question ids. As the user does not have an option to leave the question blank in a BildinBildin game, only the numbers of correct and wrong answers are collected. At each round of question, `remainingDuration` field is updated and the total number of seconds user survives the game is stored in `totalDuration` field. Records are cached within the web server and are not stored in the database. However, if need arises, they can be calculated from this collection with a simple aggregation that chooses value of maximum `totalDuration` for each user, and sorts them.

5.2.4 Tests

The structure in which tests are stored in the database changed a lot during the transition from the first version of Sınavo to the second one, to suit the needs better. As a result, a new complex structure emerged, with more usability and customizability. Question groups are introduced to the database to provide more suitable hierarchical structure, with the following schema.

```
questiongroups : {
  name : String,
  testType : String,
  questionCount : Number,
  firstIndex : Number,
  lastIndex : Number,
  subjects : [{
    subject : { type : Number, ref:"Subject"} ,
    firstIndex : Number,
    lastIndex : Number
  }]
}
```

Such a document stores the name of the question group, the type of test in which this question group appears, as well as some default values. These default values include the rank of the first question and the last question that belong to this question group, in a test. As tests consist of several question groups, each question group starts and ends at some specific part of a test. For example, a test with two question groups, such as Mathematics-2 and Geometry, questions that belong to Mathematics-2 group can start with the first question of the test and continue to a rank, let's say n. Then, Geometry would start at $n + 1st$ question and continue to the end of the test. In addition, each question group consist of one or more subjects, that generate question groups similar to the way question groups generate tests. Hence, a question group stores an array of sub-documents with a reference to the subject, and the first and last indexes of questions that belong to that subject. Note that these are merely the default values, as the same type of tests usually follow the same structure, but not always. Real values, specific to a test instance, are stored in test documents which follow the structure given below.

```
tests : {
  _id : ObjectId,
  name : String,
  startDate : Date,
  endDate : Date,
  duration : Number,
  type : String,
  solveCount : Number,
  questionCount : Number,
  availableToPublic : Boolean,
  resultAnnounced : Boolean,
  availableAfterEndDate : Boolean,
  configuration : [{
    questionGroup :
      {type : ObjectId, ref : "QuestionGroups"},
    firstIndex : Number,
    lastIndex : Number,
    subjects : [{
      subject : {type : Number, ref : "Subject"},
      firstIndex : Number,
      lastIndex : Number,
      questions : [{
        no : Number,
        question : {type : Number , ref : "Question"}
                }]
            }]
        }],
}
```

Field `configuration` stores the actual configuration of how question groups, subjects, and questions form the test. On the top level, an array of sub-documents regarding question groups is stored, with actual first and last index of questions. Within these sub-documents, subjects are stored with their first and last indexes with the constraint that the intervals of subjects [`firstIndex`, `lastIndex`] are included in the interval given by the `questionGroup`. However, this constraint is not applied to the database. It is developers' responsibility to ensure that there is no violation. Subject configurations store an array of question id-no pairs, that stores at what rank each question will be. Note that although storing only question id-index pairs might be enough to generate a test, generating complex reports based on question groups or subjects require the extra data.

When a user finishes solving a test, Sınavo system generates many documents that summarize the performance of the user on that test. These include partly performances on subjects, question groups, and the overall performance of the user on that test. Statistics on a specific subject are stored within `testSubjectStats` collection with the following schema.

```
testsubjectstats : {
  user : {type: ObjectId, ref: "User"},
  test : {type: ObjectId, ref: "Test"},
  subject : {type : Number, ref : "Subject"},
  wrong : Number,
  correct : Number,
  empty : Number,
  total : Number,
  duration : Number,
  net : Number,
}
```

These test subject statistics documents come in handy especially when all users are required to be ranked by their performances on a subject. To produce such a ranked list, fetching all documents in `testsubjectstats` collection and ordering them by net field is enough. With a similar concern, question group statistics of users are also summarized and stored within `questiongroupstats` collection with the following schema.

```
questiongroupstats : {
  user : {type: ObjectId, ref: "User"},
  test : {type: ObjectId, ref: "Test"},
  questionGroup : {type : ObjectId, ref : "
     QuestionGroups"},
  wrong : Number,
  correct : Number,
  empty : Number,
  total : Number,
  duration : Number,
  net : Number,
  subjectStats : [ {type : ObjectId, ref : "
     SubjectStats"}]
}
```

The only difference these documents have over documents holding subject statistics is that as question group statistics consist of subject statistics, these documents store a list of references to subject statistics, which come in handy when fetching both question group statistics and subject statistics at the same time, instead of querying `testsubjectstats` collection. The most general summary of a user's performance on a test, is stored within the `testresults` collection with the following schema.

```
testresults : {
  user : {type: ObjectId, ref: "User"},
  test : {type: ObjectId, ref: "Test"},
  wrong : Number,
  correct : Number,
  empty : Number,
  total : Number,
  duration : Number,
  net : Number,
  questiongroupstats : [ {type : ObjectId, ref : "
     QuestionGroupStats"}]
}
```

Similarly to question group statistics documents, these documents store an array of references to question group statistics, in addition to the overall performance of user on the specified test. As a result, overall statistics of a user on a test and the specific statistics on question groups and subjects form a tree structure, which can easily be followed by the references stored in parent objects.

5.2.5 Statistics

The second version of Sınavo also stores incremental statistics of users, with the same motivation as discussed in relational database section. However, because the main idea behind document based storage is to store all relevant data in a single document, statistics of questions are not stored in different collections, but rather within question documents. Hence, only users' monthly and overall performances are stored in separate collections with following schemas.

```
statspersubject : {
  user : {type: ObjectId, ref: "User"},
  subject : {type: Number, ref: "Subject"},
  wrong : Number,
  correct : Number,
  empty : Number,
  total : Number,
  duration : Number
}
```

```
statspersubjectmonthly : {
  user : {type: ObjectId, ref: "User"},
  subject : {type: Number, ref: "Subject"},
  wrong : Number,
  correct : Number,
  empty : Number,
  total : Number,
  duration : Number,
  month : Number,
  year : Number
}
```

The only difference between the documents stored in these two collections is that `statspersubjectmonthly` stores statistics for specific month; whereas, `statspersubject` stores overall performances. As data replication is not deemed to be problematic, redundant documents are again used for performance optimization purposes.

5.3 Graph Database

Next version of Sınavo is planned to work on a graph database.

Graph database structure was designed from scratch unlike the document based database structure, which resembles relational database in some ways that affects its design. However, designing a graph database is fairly intuitive with some exceptional cases. Designing process is pretty straightforward. All the entities in the real world corresponds to vertices and all relationships among them are denoted with edges between those vertices.

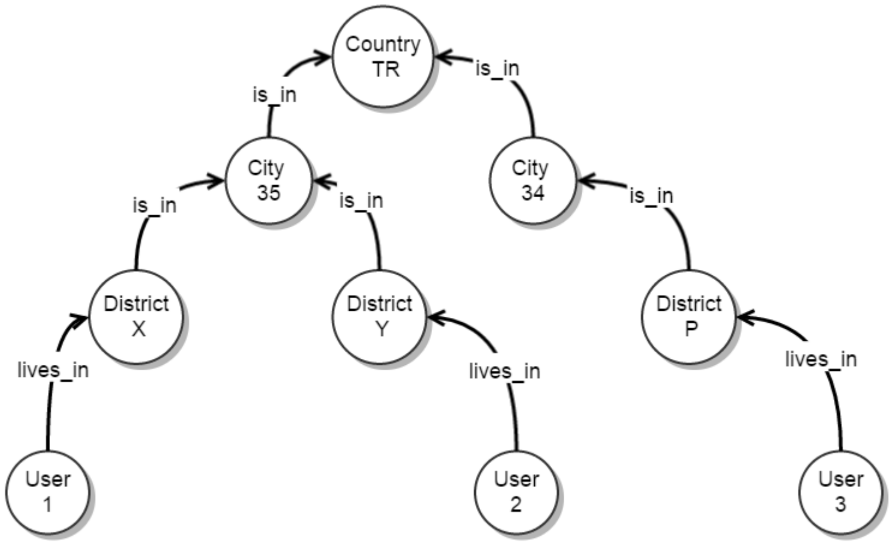
5.3.1 User data & Interactions

Vertices and relationships in Neo4j database store properties as key-value pairs. Note that Neo4j supports semi-structured data. So, the structures denoted in this section are not to be strictly implemented by all entities or relations.

User nodes will have the following key-value pairs as their properties.

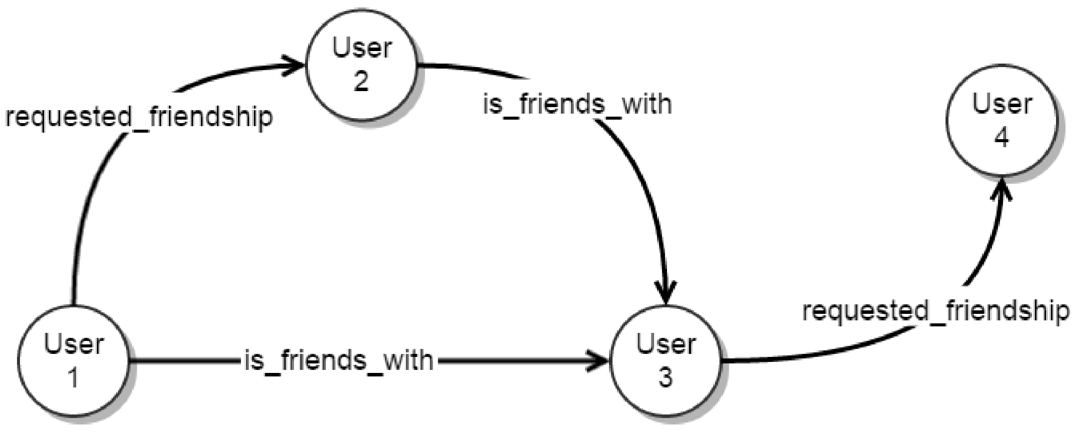
```
_id : ObjectId,
emailValidated : Boolean,
username : String,
fullname :String,
email : String,
password : String,
facebookId: Number,
facebookEmail: String,
isAdmin : Boolean,
lastLogin : Date,
dateJoined : Date,
telephone : String,
isBanned : Boolean,
```

All the properties removed from MongoDB documents will have corresponding relationships. For example, city and district the user lives in are stored within a user document in MongoDB. In the graph database; however, cities and districts are entities and are represented by separate nodes. As a result, locations of users will be denoted by edges between user nodes and district nodes as given below.



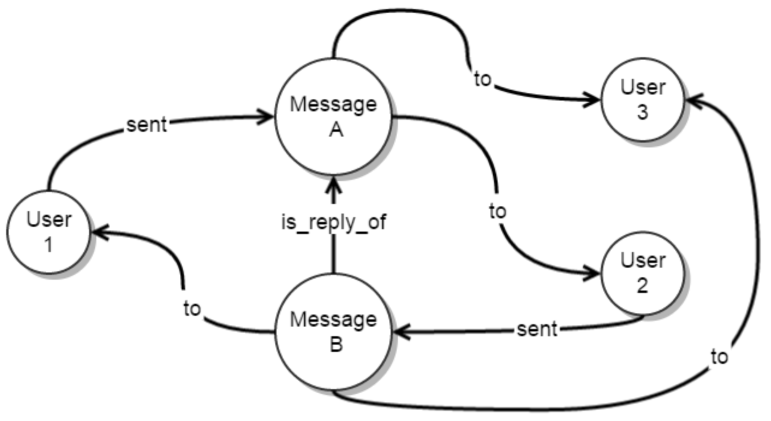
The district and the city in which a user lives in can be found by following `lives_in` and `is_in` relations emanating from user nodes. Any details required regarding the address of the user can be embedded into `lives_in` as a property. Users that share same district or city can easily be found by following those relations backwards to users.

Friendship relations are perfect fit for the power of graph database design. Phases of friendships, such as a user requesting a friendship from a user, or a user declining a friendship request are just simple relationships among user nodes.



When a user requests friendship from another user, a `requested_friendship` relationship is created between them. The date at which this request occurred is embedded into the relationship as a property, along with a flag that denotes if this request is declined or not. If a user declines a friendship request, this flag is set to true. If the user decides to accept this request, `requested_friendship` relationship is deleted and `is_friends_with` relationship is created instead, the direction of which differentiates the one who initiated the friendship. To find friends of a user is as easy as tracing all `is_friends_with` relationships.

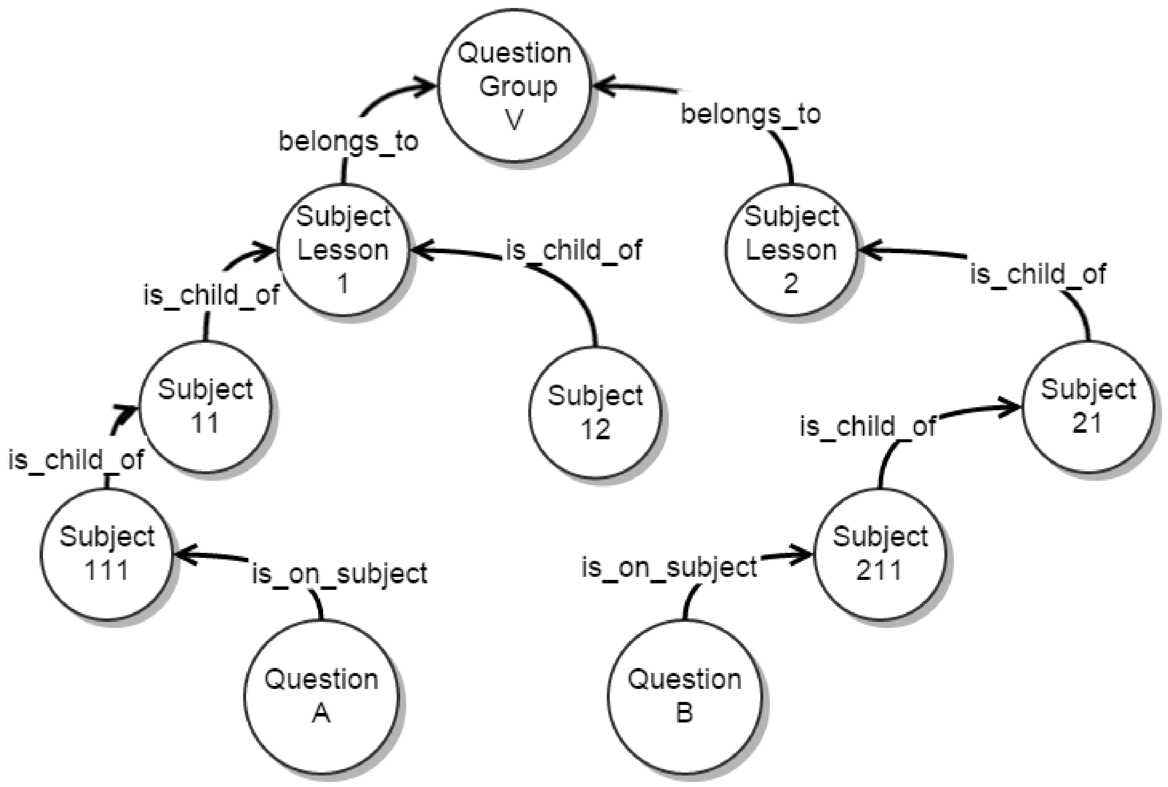
Complex messaging structures are easy to store in a graph database. These structures can include threads, multiple receivers, and replying to messages other than the most-recent one. Such a messaging schema can be seen in the graph below.



A string of messages starts by a user sending a message to a set of users. In the example given above, "User 1" initiates the messaging by sending "Message A" to both "User 2" and "User 3". The date and the text of message are stored as properties of the message node. The relationship `to` denotes the recipients of that message, and within this relationship, a flag `seen` is stored that shows if the recipient has seen the message or not. Any subsequent messages, such as "Message B" will be linked to the first message via a `is_reply_of` relationship, again with links to users that send and receive it. A string of messages can be populated by first finding a message node without any `is_reply_of` relationship emanating from it, and traversing `is_reply_of` relationships directed to that or any subsequent message nodes.

5.3.2 Questions - Subjects

As graph databases are designed to handle complex relationships, subject tree and question groups mentioned earlier are very suitable to be stored in one. Nodes representing question groups will store only their names. Within the relationships `belongs_to` that leads to question group nodes will include default values of first and last indexes of questions that are included in that question group, as mentioned earlier.



Subjects that are at the first level of the subject tree are also labelled as lessons. This eases the process of locating only the lessons, instead of searching for subject nodes that do not have a parent. Each child of a subject will be connected to it with a `is_child_of` relation. Subjects have the following properties.

```
_id: Number,
name: String,
numberOfQuestions : Number
```

Question nodes are only linked to leaves of the subject tree, as it is easy to traverse the tree upwards to locate any non-leaf subjects. Questions will have the following properties.

```
_id : type : Number,
answer : String,
qx : Number, qy : Number,
ax : Number, ay : Number,
bx : Number, by : Number,
cx : Number, cy : Number,
dx : Number, dy : Number,
ex : Number, ey : Number,
total : Number,
correct : Number,
wrong : Number,
empty : Number,
duration : Number,
durationSquared : Number,
isChecked : Boolean,
isFaulty : Boolean,
difficulty : [Number]
```

The only difference between questions stored in MongoDB and Neo4j is that no references are kept to subjects as relationships replace references in a graph database. All other properties stand for the same purposes as mentioned in section 5.2.2.

Solved questions are not represented as vertices in Neo4j as they are merely relationships between users and questions with some additional data. Hence, every time a user solves a question, a new relationship, called `solved`, is created between the user and the question. All related data is stored within this relation's properties, with the following schema.

```
date : Date,
answer :
  {type: String, enum: ["A","B","C","D","E","X"]},
duration : Number,
result :
  {type: String, enum: ["correct", "wrong", "empty"]
```

Note that unlike relational database and document based database, in these relationships only store acts of solving questions that are not during a test or a game. Those acts are mentioned in the following sections.

5.3.3 Games

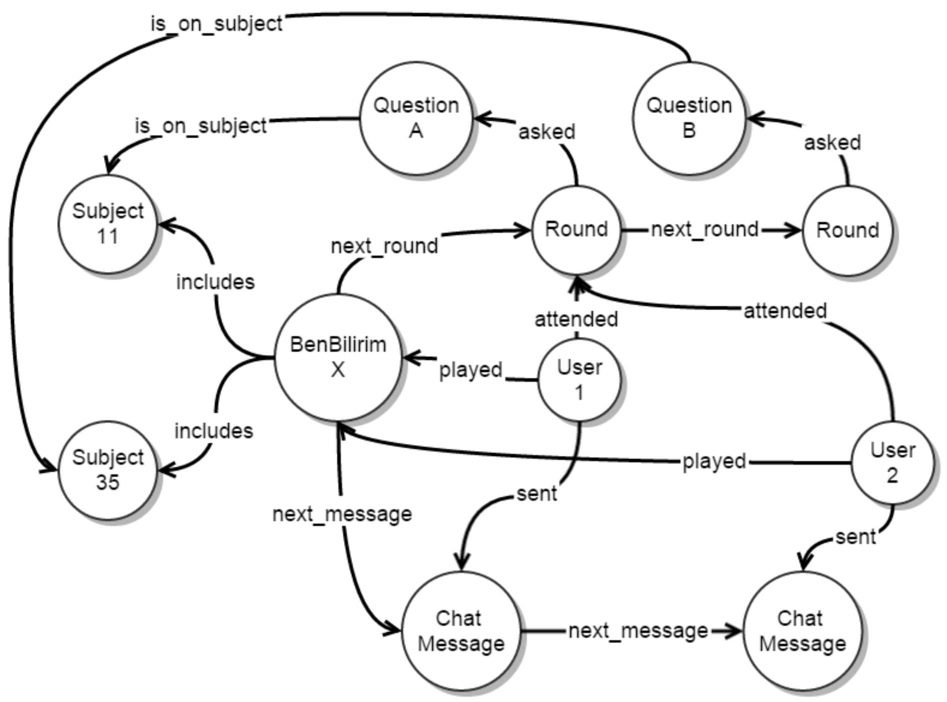
Ben Bilirim games require a complex graph to be represented, as there are many aspects to this kind of game. A node that represents a Ben Bilirim game will have the following properties.

```
_id : Number
name : String,
createdDate : Date,
maxErrors : Number,
state :
  { type: String , enum: ["NotStarted","RoundBreak",
     "Running", "Finished"], default: "NotStarted" },
currentRound : Number,
startDate : type: Date,
```

Again, these properties are the same as in the document on MongoDB, excluding references. For example, subjects chosen at the creation of a game are referenced via `includes` relationship between the game node and a set of subject nodes, without any properties in them. Users, on the other hand, are connected to the game node with relationships `played`, which holds the results of the game concerning the connected user as properties, including the round the user eliminated and the rank of the user.

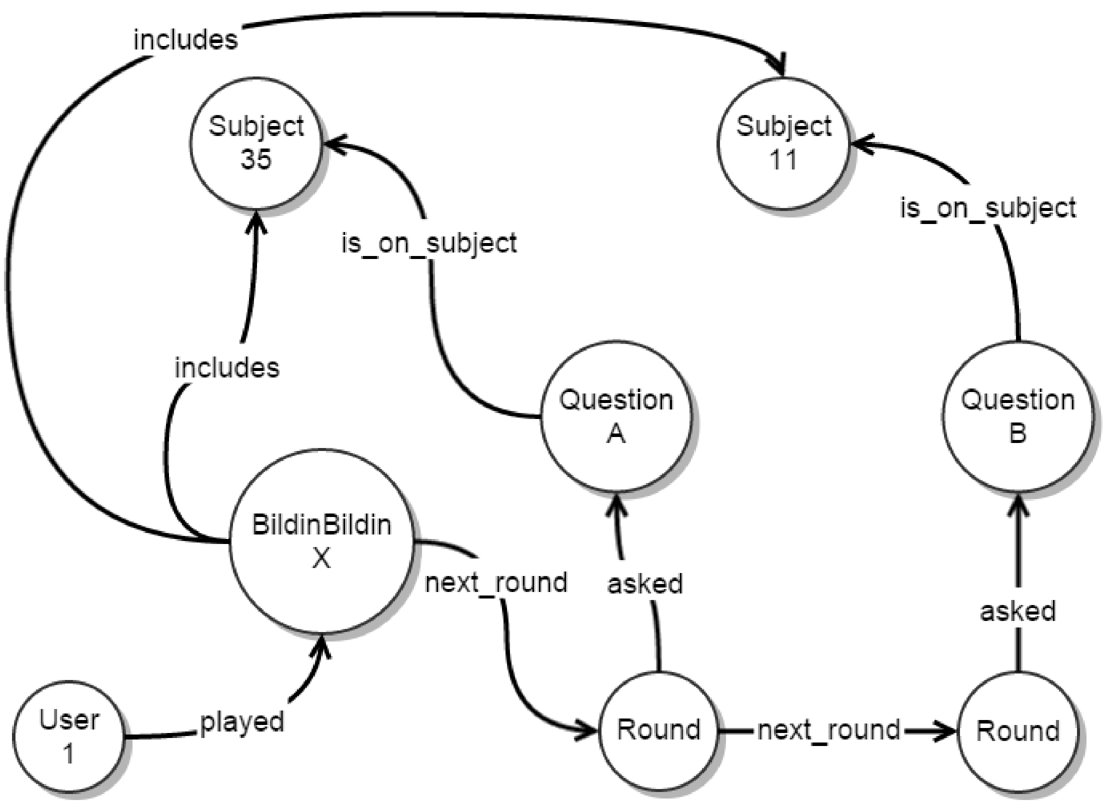
For each round played during the game, a round node is created with a relationship to the question asked during that round. As rounds are sequential, there is a relationship called `next_round` between each consecutive round. As the first round does not have any preceding round, this relationship emanates from the node that represents the game. To simplify any queries that traverse through these sequential rounds, the relationship that bounds the game and the first round is also named `next_round`.

Users that contested in a round are also connected to the node which stands for that round with a relationship called `attended`. This relationship includes the answer user chose, and the result enumerated as correct wrong or empty.



Each chat message sent during a game is represented with a node labelled "Chat Message". Again, as chat messages are sequential, each node is connected to the next with `next_message` relationship. User that sent the message is also connected with the chat message node with a relation `sent` that holds the date at which the message was sent as a property.

Bildinbildin games have similar but simpler graphs compared to BenBilirim games.



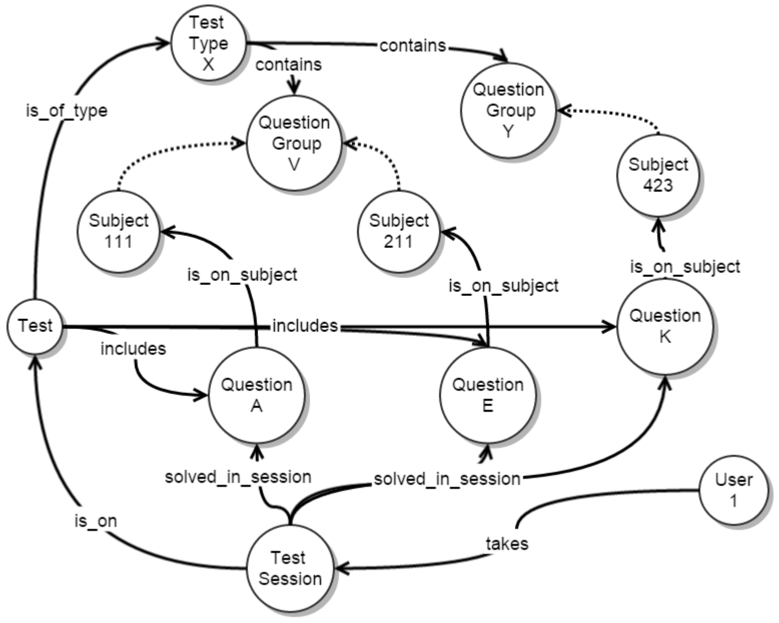
As there are only one user that plays the game, the user is connected to the BildinBildin node with a `played` relation. Details of the game is stored as properties of the game node with the following schema.

```
bildinbildins : {
  createdDate : Date,
  remainingDuration : Number,
  totalDuration : Number,
  correct : Number,
  wrong : Number,
}
```

The game is connected to a string of nodes representing rounds. Each round is also connected to a question node with `asked` relationship. Details of a round include the answer use chose, the result of that round and the number of seconds the round took.

5.3.4 Tests

Tests require a complex graph with many relationships to existing nodes, a simple part of which is given below. Note that the relationships from subjects to question groups (denoted with dashed lines) are skipped in this graph for the sake of simplicity, as they are handled in detail earlier.



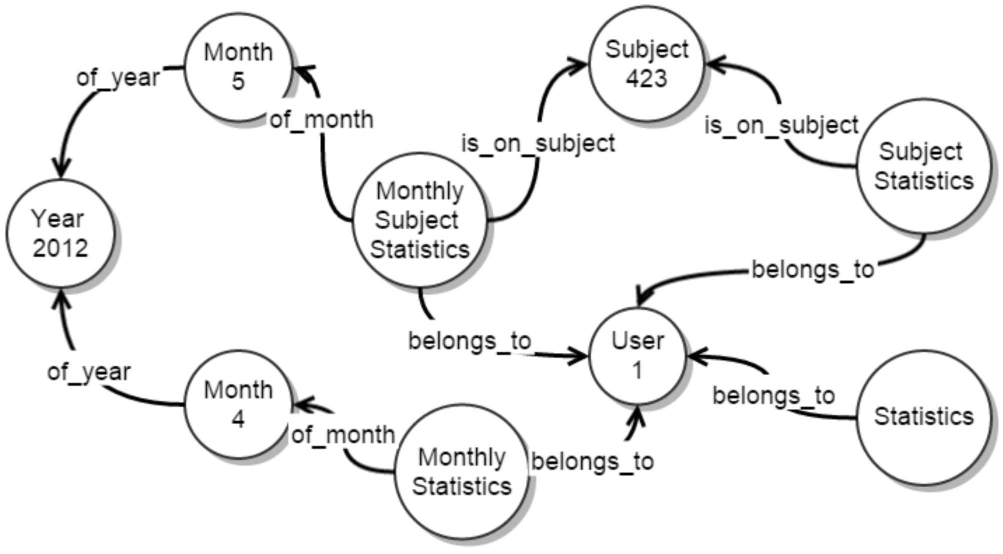
Tests can be seen as a sequence of questions. Hence, the main part of a test graph is a set of relationships among a test node with questions included in that test. The type of a test can be found with tracing `is_of_type` relationship that emanates from the node that represents the test. Note that although test type nodes are connected to question group nodes with `contains` relationship that stores the default number of questions from that specific question group as a property, tests may include different number of questions from those question groups. Properties of test nodes include the following.

```
_id : Number,
name : String,
startDate : Date,
endDate : Date,
duration : Number,
solveCount : Number,
questionCount : Number,
availableToPublic : Boolean,
resultAnnounced : Boolean,
availableAfterEndDate : Boolean
```

Whenever a user solves a test, a node with label "Test Session" is created that stores user's details on that test. This node is connected with the questions included in the test with `solved_in_session` relationship, which also holds user's answers and the seconds user spent on that question. To calculate the user's performance, all these relations are scanned and aggregated, which is then stored in the test session node as a summary.

5.3.5 Statistics

Incremental statistics are stored in graph database as the following graph.



Nodes that represent monthly statistics are connected to month nodes which are connected to year nodes. Even though it is possible to store month and year data as properties of statistics nodes, this is a design choice to find statistics regarding same month easier. Nodes that store statistics on a specific subject are connected with subject nodes with `is_on_subject` relationships. All nodes storing statistics, such as ”Monthly Statistics”, ”Monthly Subject Statistics”, ”Subject Statistics”, and ”Statistics”, share the same structure for their properties, which is the following.

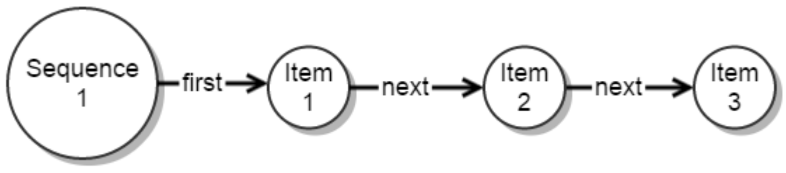
```
wrong : Number,
correct : Number,
empty : Number,
total : Number,
duration : Number
```

Note that the statistics nodes can be connected to the questions involved in these statistics. However, this will create a huge amount of redundant relationships, and is not implemented due to not being needed for Sınavo operations.

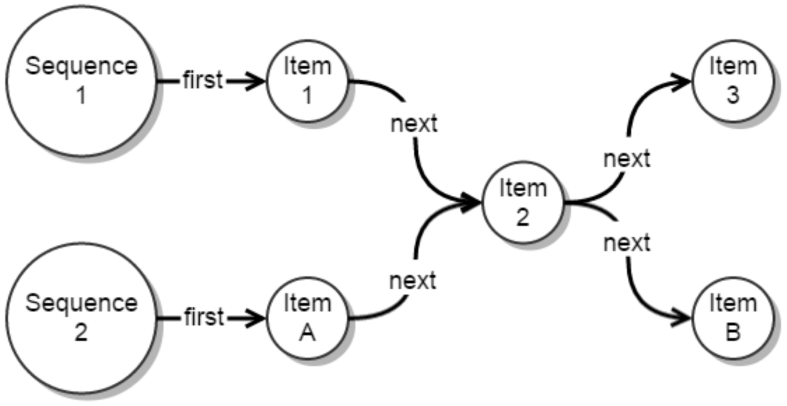
5.3.6 Design Notes

Designing a graph database is a relatively new experience to developers, especially when compared to relational schema which have been around for decades now. Here are a few patterns encountered during this design process.

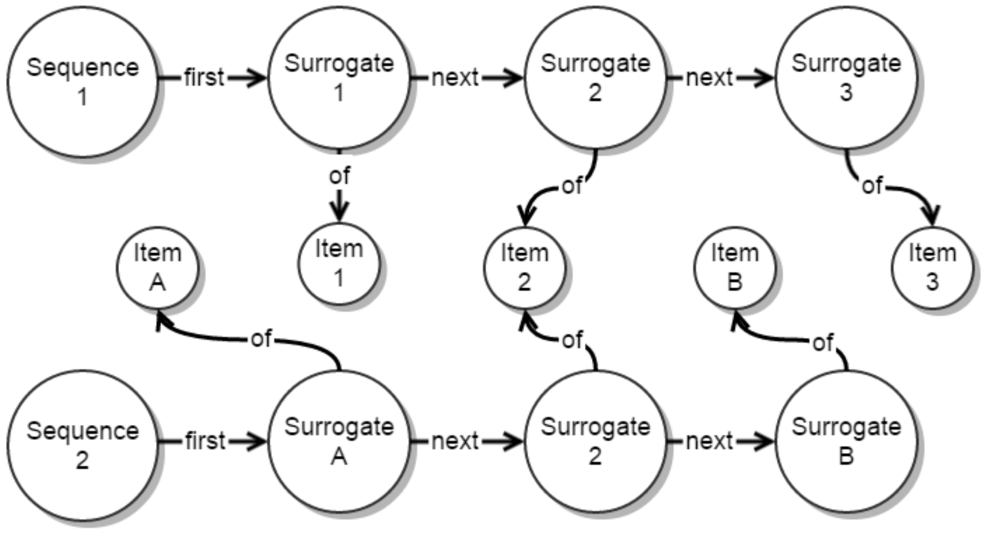
Initial instinct when generating a graph to store sequential items is to connect each item with a simple `next` relationship. Rounds in games, or questions in tests are examples for that. A simple sequence of items can be seen in the graph below.



However, when these entities can occur in more than one sequence, such as the questions in Sınavo being used in more than one scenarios, using a simple `next` relationship is not a viable option. While traversing through the sequence, there will be a node that has two `next` relationships emanating from it. Based on the sequence being traversed, which next is going to be used can be ambiguous. Consider the example given below. Sequence 1 and 2 share item 2. However, while traversing a specific sequence, it is not clear whether item 3, or item b comes after item 2.



To avoid this ambiguity, there are two patterns to be followed. It is possible to store an identifier of the sequence within these relations. Hence, when traversing a string of nodes, query should always continue with the relation that holds the correct id of the sequence it is traversing. Another possibility is to use of surrogate nodes. In this pattern, for each item used in a sequence, a surrogate node is created and `next` relationships are formed within these surrogates. Then, items that are actually within the sequence are bound to surrogates with relationships. Hence, the sequence is always followed with a `next` relationship, and when an item in the sequence is required to be located, the surrogate is found first and then the relationship `of` is traversed. The graph given below is an example of two sequences with shared items using surrogacy.



5.4 Comparison & Analysis

We have specified characteristics of three different database types. In the following sections, we analyse attributes of these database types and provide a comparison among them. We deliver these comparisons in two parts: quantitative and non-quantitative attributes. Note that even though we support our claims with examples, non-quantitative attributes might be subjective to developers. Some developers may find it very easy to design a relational database because they are used to it, although it may require a lot more relations and include many constraints.

5.4.1 Non-quantitative Attributes

5.4.1.1 Ease of Use

We deem attribute *ease of use* as both the difficulty of using a database, and the effort it takes to model a dataset in that database model. From this perspective, relational databases have one advantage: they have been around for decades. Developers are more familiar with relational databases than the ones driving NoSQL movement. For this reason, many would feel that relational databases are easier to use. However, it is obvious that relational schemas do not feel as natural as others. We have shown that to store attributes of users, the relational database requires more than four tables, if the users have Facebook accounts linked to their Sınavo accounts. Whenever data of a user is to be fetched, these tables are required to be joined. In addition, in the modelling process of a software, first an ER (Entity-Relationship) diagram is used to clarify entities. When using a relational database, only after a design process can these entities fit to a relational schema, as they do not usually comply with natural way developers denote entities. Within this design process, many rules and constraints are required to be followed to avoid normalization issues. Hence, it is safe to argue that relational databases are relatively more difficult to design and use.

When designing a dataset as documents to be used in a document based database, a similar approach is used to the one for relational databases. However, as data is denormalized on purpose, there is no normalization concerns. Each entity in an ER diagram corresponds to a document. Relations are embedded into documents, such as friendship relation among users, mentioned in section 5.2.1. Actions, such as solving questions, are stored as documents as well. As more data is embedded into a single document, there are less collections than there are relations in a relational database. In projects such as Sınavo, the number of tables in a relational database can grow to scales that are difficult to handle. As a result, document stores can be tidier and easier to use. However, the lack of inter-collection querying requires such queries to be split into parts. Each part, concerning only one collection are run separately, and then the results are merged to generate the output. Although some query parts can be run in parallel, some will require to be run in series, i.e. some queries will be run after getting results of others. This creates an excess of communication overhead with the database server. In configurations where this communication overhead is a bottleneck, the lack of inter-collection querying would cause an important problem.

Graph databases are obviously the easiest when matching the modelling of a software to a database schema. As graphs are mostly used when modelling entities and relationships among them, converting these models to a graph database usually requires minimal work. In addition, graph paradigm enables querying relations of varying length. For example, users are connected with friendship relations, in all three database types we have discussed. To be able to find a set of users U that are n friendship relationships away to a user u_1, i.e. friends of the friends of u_1 if $n = 2$, both relational and document based databases require recursive or iterative calls. However, graph databases are built for this kind of queries and can query relations of various lengths with ease. Moreover, Neo4j provides a shortest path query, using Dijkstra's algorithm [11]. If necessary, implementing this algorithm in a relational or document based database would require a lot more work.

5.4.1.2 Maintenance

From *maintenance* point of view, relational databases are solid. The difficulties they present in modelling process are design choices to avoid maintenance problems. For example, normalization rules given in section 3.2.2 are generated to avoid inconsistencies in database. If the database complies with all normal forms, there will be no data redundancy, and hence no inconsistency among copies of the same data. In addition, constraints can be introduced to avoid dangling tuples. For example, if a tuple t_1 references another tuple t_2 in a different relation, and t_2 is removed from the database, a constraint can automatically update the reference and either delete t_1 or set its reference to t_2 as null. Applying all these rules and constraints will ensure that data integrity will be held at all times.

Document stores, on the other hand, are schema free, and data is denormalized on purpose. Although this enables easier storage and retrieval, it also leads to possible inconsistencies. This approach delegates the responsibility of data integrity to the application that uses the database management system. So, developers of the application need to follow any clean-up required after updating the database. In addition, there is no guarantee that a document fetched from database will have all the expected attributes. As a result, the application running on the document store must expect such irregularities and be able to act on them. In summary, database management system is not concerned with any rules of data it stores. This is a design choice and a trade-off between ease of use and supporting maintenance, where document based databases favour ease of use.

Graph databases show similarities to document based databases when it comes to maintenance of the database. As properties of vertices and edges are key-value pairs, and data is schema free, database management system does not care about data integrity. However, Neo4j supports transaction event handlers that are hooked to transaction events such as creating new nodes or updating data of a node/relation. These event handlers can be used, as triggers in relational databases, to ensure data consistency after dataset is updated. Transaction events are fired both before and after data update. When event handlers are hooked to events, they notify “beforeCommit” or “afterCommit”, which denotes that handler will be called before any changes are committed in the database, or after.

5.4.1.3 Flexibility

All three database paradigms have similar expressive power, that is, all three data models can represent any dataset. However, with a strict schema, expressing same dataset with a relational model can require more work. This work can feel cumbersome especially when it comes to extending an already existing database with new properties and/or entities. Even small changes might lead to introducing multiple new tables. In addition, extending relations with new attributes requires careful handling. Tuples that already exist in database require default values for new attributes, as they are bound to follow the schema as it is.

Both document based databases and graph databases, on the other hand, are more flexible to be extended. As there is no single schema being followed, data is expected to have nulls. This way of thinking comes in handy especially when new attributes are required for existing data. If the application is robust in handling missing properties, there is no excess work required. In case of new entities being introduced into the database, data model of document store is more suitable, because collections are designed to encapsulate similar data and inter-collection relations are not very common. Hence, modular structure of collections do not require much work when independent documents are changed. Moreover, as data is stored as JSON objects, almost any kind of data structure can easily be stored within these documents, including subdocuments. However, Neo4j requires data on nodes and edges to be in key-value pairs, which is less flexible.

5.4.2 Quantitative Attributes

The most obvious quantitative attribute of a database is its speed in accessing data. We have conducted experiments on Sınavo data stored in these three databases. With different types of queries, we have tried to capture how these databases handle queries of different patterns. We have collected query execution times in milliseconds and present results in this section. Some operations require more than one query submitted to the database management engine. For such queries, we start measuring the time before the first query, until right after the last query. Each operation is executed 1000 times from a Node.js application using lightweight drivers to avoid added time caused by driver overhead. We have hooked time measuring operations into the drivers to avoid adding time it takes to adjust data for the application after query is executed.

We have used Microsoft SQL 2008 R2 as relational database, and query the database using mssql module of Node.js. We have used MongoDB 2.4.8 and connect to it using official *mongodb* module for Node.js. For graph database, we have used Neo4j Community 2.1.6 and used *neo4j* module within Node.js. The experiment was done on a PC with Intel Core i7-3630QM 2.40 GHz CPU with 8 gigabytes of ram and 64 bit Windows 8.1 as operating system.

5.4.2.1 Query with fixed number of joins

The first query retrieves the subjects of the last three questions solved by a user. As there are three layers in the subject tree and questions belong to the subjects in the lowest layer, each question is bound to three subjects. Maximum, minimum and averages are gathered over 1000 executions and displayed in Figure 5.2.

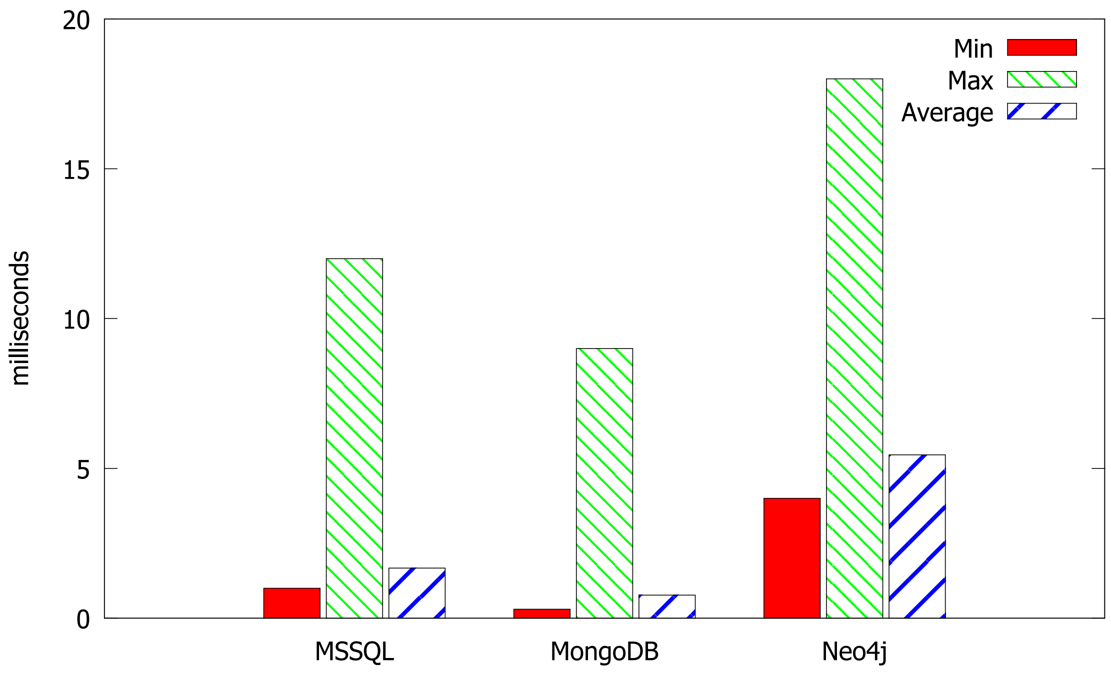


Figure 5.2: Execution durations for query that retrieves the subjects of the last three questions a user has solved

In SQL, this query requires to locate the last three tuples in `TBL_QuestionSolved`, find relevant questions and join them with `TBL_Subjects` to find the subjects. As there are three subjects to be found, `TBL_Subjects` needs to be joined three times. The following query gets the operation done.

```
SELECT S1.Name, S2.Name, S3.Name
FROM TBL_Subjects S1, TBL_Subjects S2,
  TBL_Subjects S3, TBL_Question Q
WHERE S1.Id = Q.SubjectId AND S1.ParentId = S2.Id AND
    S2.ParentId = S3.Id AND Q.QuestionId IN
  (SELECT TOP 3  QS.QuestionId
   FROM TBL_QuestionSolved QS
   WHERE QS.UserId = {UserId} ORDER BY QS.Date DESC)
```

As MongoDB can query only one collection at a time, this operation requires separate calls to database management system. First, last three documents in `solvedquestions` collection are found. Then, only call required is finding subjects in `subjects` collection, as subjects are already stored in solved question document due to denormalization of data. Following queries are run in order to gather required data.

```
    db.solvedquestions.find({user : {userId}})
      .sort({date : -1}).limit(3);
    db.subjects.find({_id : {$in : {subjectIds}}});
```

Cypher query used to collect the required data is relatively simple. It firstly locates the user node. Following `solved` relation emanating from the user node, questions are found. After that, leaf node on the subject tree is found by following `is_on_subject` relation, and other subjects via `is_child_of` relations. The following cypher query is required to collect required data.

```
MATCH (u:user {id : {userId}})-[r:solved]
  -(q:question)
WITH q
ORDER BY r.date DESC
LIMIT 3
MATCH q-[:is_on_subject]-(s)-[:is_child_of*]->(s2)
RETURN s, s2
```

5.4.2.2 Authentication query

The second experiment conducted on these databases is an actual case in Sınavo system. Queries run in this experiment take an email address and a hashed password as input, and try to locate the user account with these information. This work is being done regularly by the authentication system of Sınavo. The details of the user are returned by this query, including the user's name and surname, email address, the city and the town the user resides in, along with the possible Facebook account details, if the user has it linked with their Sınavo

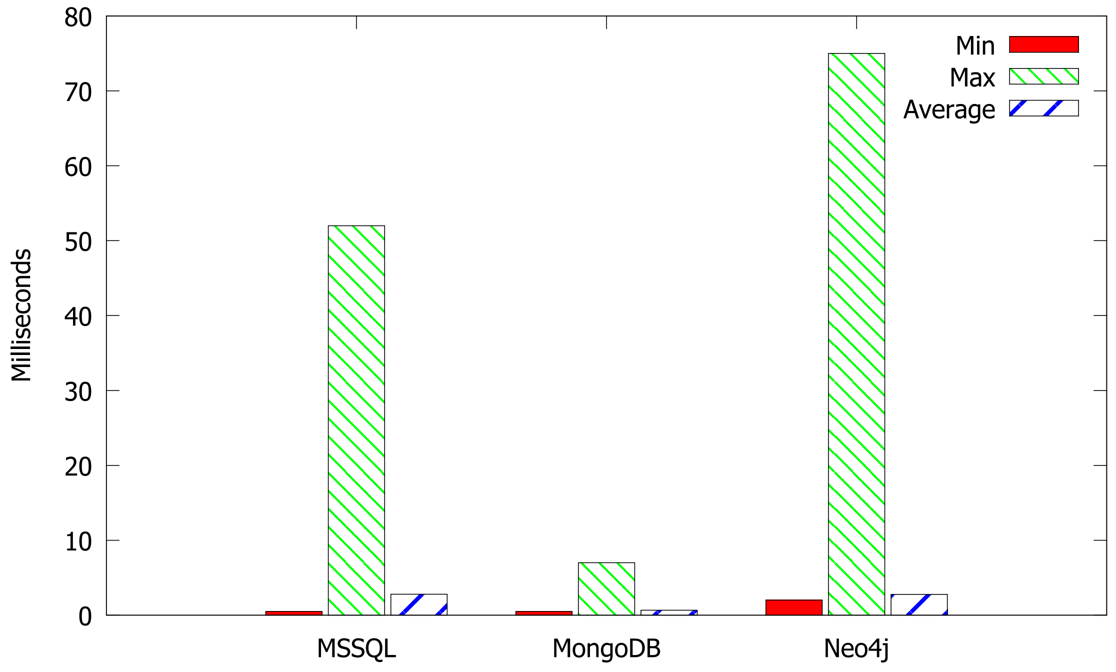


Figure 5.3: Execution durations for authentication query

account. If no user can be found with the specified email address - password pair, an empty result set is returned to indicate that the user has failed to authenticate.

SQL version of this query requires locating the tuple with the given input in `TBL_Auth` table, and get other details about the user from other tables. These details include general details about user that reside in `TBL_UserInfo` table, which is matched with the tuple in the `TBL_Auth` table over `UserId` field. A reference to the town the user lives in is stored in `TBL_UserInfo` table as field `TownId`. Using this reference, first the town is located in `TBL_Town` table, and using data from this table, city name is retrieved from `TBL_City` table. As it is possible for users not to have corresponding Facebook accounts stored in Sınavo database, we use an outer join to join `TBL_FacebookAuth` with `TBL_Auth`. Outer join retrieves the tuple from `TBL_Facebook` if there is one with the given `UserId`, and returns `NULL` otherwise. If outer join is not used to join these two tables, the result set will be empty in the case of users without Facebook account data. The following query is used to authenticate users in MSSQL.

```
SELECT UI.BirthDate, UI.TownId, A.*, FA.FacebookId,
  FA.FacebookEmail, T.Name, C.CityName
FROM TBL_UserInfo UI, TBL_City C, TBL_Town T,
  TBL_Auth AS A LEFT OUTER JOIN TBL_FacebookAuth FA
  ON A.UserId = FA.UserId
WHERE UI.UserId = A.UserId AND T.Id = UI.TownId AND
  T.CityId = C.Id AND A.Email = {email} AND
  A.Password = {password}
```

Because most of the details of users are embedded into user documents in MongoDB, authentication query requires less joins. Main part of this query is to locate the user document that has matching email address and password pair with the input. As cities and districts are kept separately, with references to them in user documents, retrieving the names of the city and the district the user resides in requires the user document to be joined with city and district documents. Because of the lack of inter collection queries in MongoDB, Only after the user document is located can another call be made to retrieve the city and the district. The following queries handles authentication in MongoDB.

```
    db.users.find({email : {email},
                   password : {password}});
    db.cities.find({_id : {user.city}});
    db.districts.find({_id : {user.district}});
```

The authentication query in Cypher is pretty straight-forward. Database engine locates the user node with the given email address and password pair, and follows `lives_in` relationship from the user node to locate the district, and `is_in` relationship from the district node to locate the city. The following query handles this job in Neo4j.

```
MATCH (u:user)-[:lives_in]->(d:district)
  -[:is_in]->(c:city)
WHERE u.email = {email} AND u.password = {password}
RETURN u, d, c
```

5.4.2.3 Query with different path lengths

The last experiment is designed to estimate abilities of database systems on handling relationships over different path lengths. We define path length as the number of transitions over different entities. The query retrieves a predefined number of users, 20 in our case, connected via friendship to a user with exact path length given as input. The simplest case is friends of a user u, which corresponds to a path length of 1. Friends of friends of u are connected to u with a path length of 2, and so on. We have conducted this experiment for path lengths from 1 to 5. For each path length, we have measured 1000 executions and used their averages. Results can be seen in Figure 5.4.

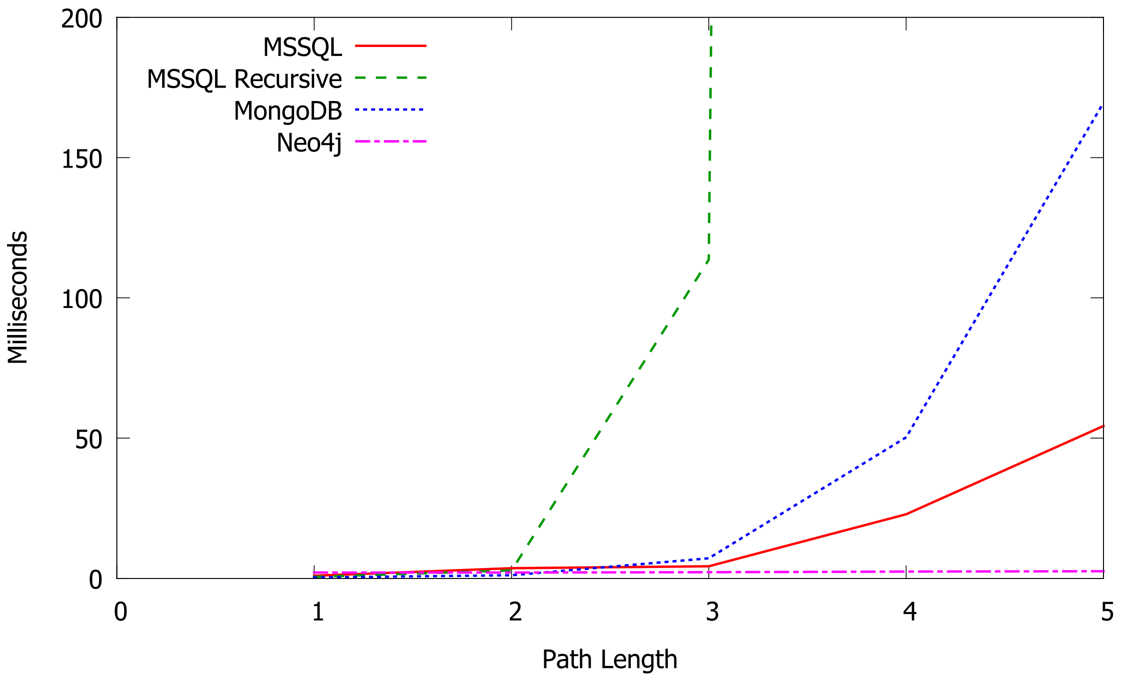


Figure 5.4: Query execution durations for different path lengths over 1000 executions

For this query, we have implemented two different approaches in MSSQL. First one is a recursive query that uses common table expressions. It first generates a view with friendships of the user including the path length as `Level`. Then, it queries that view to get the results. The following query gets the job done.

```sql
WITH FriendRec(UserId, Friend, Level)
AS
(
  SELECT UserId1 AS UserId, UserId2 AS Friend ,
    0 AS Level
  FROM TBL_Friendship F
  WHERE F.UserId = {UserId}
  UNION ALL
  SELECT UserId1 AS UserId, UserId2 AS Friend ,
    (FR.Level + 1) AS Level
  FROM TBL_Friendship F INNER JOIN FriendRec FR
    ON F.FriendId = FR.UserId
  WHERE FR.Level < {Length};
)

SELECT DISTINCT TOP 20 Friend
FROM FriendRec
WHERE Level = {Length}
```

The other approach for MSSQL require multiple calls to the database management system. Application loops from 1 to the number of path length required and each query retrieves users connected to user with path length of the index of the loop. For path length 5, this approach requires 5 separate calls to database management system, each using output of the previous one. The following query is the one that is submitted multiple times:

```sql
SELECT DISTINCT UserId2
FROM TBL_Friends
WHERE UserId IN {UserIds}
```

Note that only the last call limits the number of tuples returned to 20.

MongoDB resembles non-recursive SQL approach, as it requires more than one call to database. Each call retrieves the friends of users given as input. Again, only on the last call the results are limited to 20. The following MongoDB query is used.

```
    db.users.find({_id : {$in : {UserIds}}},
      {friendship : 1});
```

The way Neo4j handles this kind of queries is a lot simpler. With a single query, it locates the user node and traverses the graph over `is_friends_with` relationships to find users with distance given as input.

```
MATCH (n:user)-[:is_friends_with*{Length}]-(u)
WHERE n.id : {UserId}
RETURN  u.nodeId LIMIT 20
```

5.4.2.4 Comparison Summary

For the first query, it can be seen in Figure 5.2 that MongoDB outperforms other databases. Although SQL performs similar to MongoDB, denormalized data schema enables MongoDB to avoid extra joins required in SQL. As subject ids are already embedded to solved question documents, only call done is a query to subjects collection; whereas, SQL requires locating the question first, than matching the subjects three times. It is obvious that Neo4j performs worst, compared to SQL and MongoDB.

The second experiment implements the authentication system in Sınavo. The results given in Figure 5.3 clearly show that, just like in the first experiment, MongoDB performs better in this query too, with less than a millisecond on average. Although maximum execution time for Cypher is worse than all others, average execution time is almost the same with SQL.

With the third query, we investigate how different data models respond to querying relationships with varying path lengths. We have applied two approaches for relational database, one with recursion and one with iteration. We

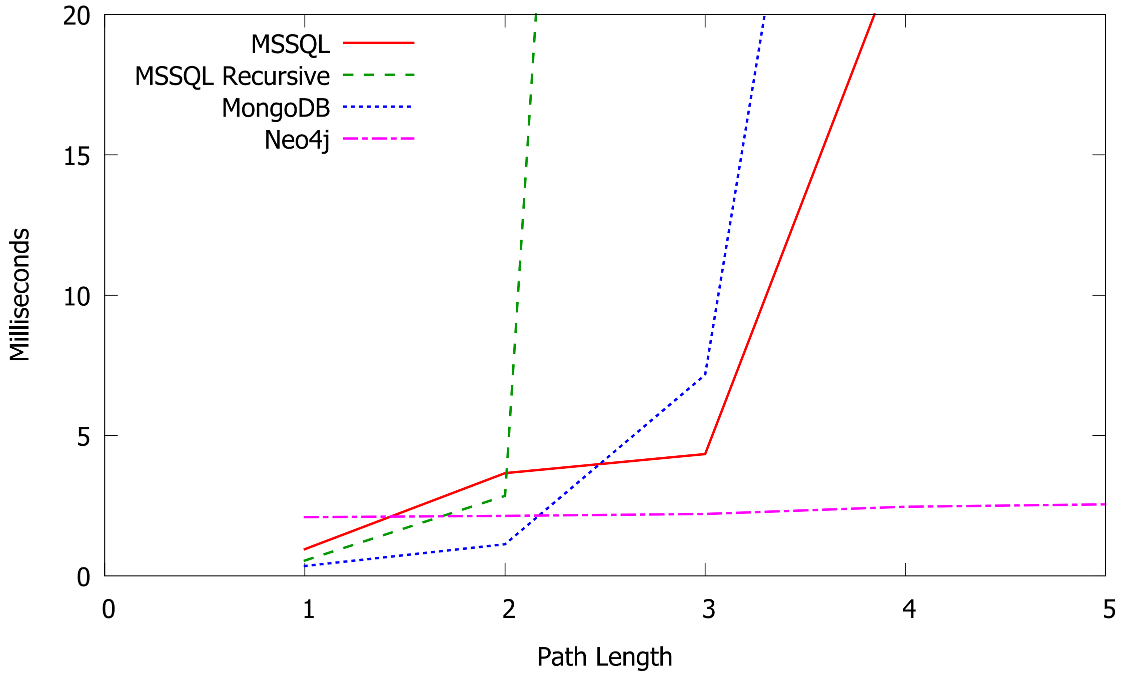


Figure 5.5: Closer look at query performances for different path lengths

have analysed how increasing path length affects query performance, and seen that Neo4j is a clear winner in this comparison, for higher path lengths. It can be seen in Figure 5.4 that Neo4j does not suffer in higher path lengths and continues to perform close to queries for lower path lengths, resulting in a horizontal line. Although other databases perform better for lower path lengths, increase in execution time for higher path lengths are worrisome. To investigate performances further on lower path lengths, closer look is given in 5.5. Recursive approach of SQL performs better than iteration for path lengths 1 and 2; however, its performance decreases tremendously after 2, resulting in max recursion errors after 3, and hitting 15 seconds request timeout limit at 5. MongoDB outperforms all others for path lengths 1 and 2, as it does for the first query. But, execution time increases vitally after 2.

To summarize, as these experiments show, databases perform differently under different types of queries. Hence, there is no one database and data model that fits all types of datasets. Instead, the ways that dataset is going to be queried is a vital determinant in the process of choosing database and data model.

5.5 Summary

We have discussed the design and implementation of Sınavo system on three database management systems. We have shown that relational database requires more tables than others; whereas, the design process for graph database is easier than others as it does not require a transition from description of the data model to a data model suitable for the database.

In section 5.4.1 we have focused on non-quantitative attributes of these three database management systems and explain how relational databases are easier to maintain. However, document-based databases and graph databases are more flexible. In section 5.4.2, we have introduced the results of experiments we have conducted and shown that although with a small or fixed number of joins MongoDB and Microsoft SQL Server perform better, for locating entities with varying or long paths, Neo4j outperforms others.

Chapter 6

Data-driven Decisions

Storing and processing Big Data is a big challenge to tackle when developing systems that undergo huge data generation. However, arguably the most important part of a Big Data system is the ways this collection of data is processed to make practical data-driven decisions.

Users of Sınavo generate a vast collection of data, as discussed earlier. There lies valuable inferences to be made from this collection regarding entities in Sınavo, from performances of students to characteristics of questions. This section describes some of those inferences implemented and how data is used to make useful decisions and classifications.

6.1 Statistics of Sınavo

As discussed earlier, core of Sınavo is users solving questions. From this simple interaction of entities, there are many practical results to be inferred. For example, performance of a user on a subject is stored as a collection of acts of solving questions on that subject. However, these statistics are only partial data and would not reflect to real world with a one-to-one mapping. A user that solves a small set of questions on a subject correctly is hardly expected to keep this success in a real world test. What needs to be done is to evaluate these statistics, and come up with ways to estimate real world performances of users based on their Sınavo performances.

In addition to this approach, there are more ways this data can be used. The act of solving a question is a bidirectional matter. As users solve a question and affect their performances, that question is being solved that sums up to statistics regarding that question, which can be used to guess how the users that are yet to solve that question would perform. This estimation of users' performances on a question can be labelled as difficulty of that question. Estimating difficulties of questions using statistical analysis can be automatically done, rather than the traditional way of hiring experts to do the job. Consequently, it is a lot cost-efficient.

The center of attention while making data-driven decisions in Sınavo is the results of users solving questions. To accommodate these results in a more mathematical way, we map these results to 0 and 1; 1 corresponding to the question being solved correctly, 0 to not a successful attempt. Leaving a question empty is regarded as solving it wrongly, as it means that the user failed to solve the question correctly. From users' perspective, these results are a collection of ones and zeroes for their successful or unsuccessful attempts on solving a set of questions. From questions' perspective, these results correspond to a collection of ones and zeroes for them being solved successfully or unsuccessfully. This approach enables us to see these acts of solving question as a Bernoulli Trial (also known as Bernoulli Experiment), which is an experiment with only two possible results: success or failure.

In such a series of Bernoulli trials where probability of resulting in success is θ, the probability of getting exactly y successes from n trials follows a discrete probability distribution which is a binomial distribution $Bin(n, \theta)$, and can be calculated as given below,.

$$Pr(X = y) = \binom{n}{y}\theta^y \times (1 - \theta)^{n-y}$$

Although finding the real value of θ is intractable, what we are interested in is to estimate the value of θ up to a point. To make this estimation, we have a subset of actual values. From a question's perspective, the actual set of results of these Bernoulli trials would be the array of ones and zeroes formed when all users, current and future, solves that question. However, it is not feasible to expect getting all of these actual data. Instead, we do statistical analysis on the subset of results we have, and guess what θ is likely to be.

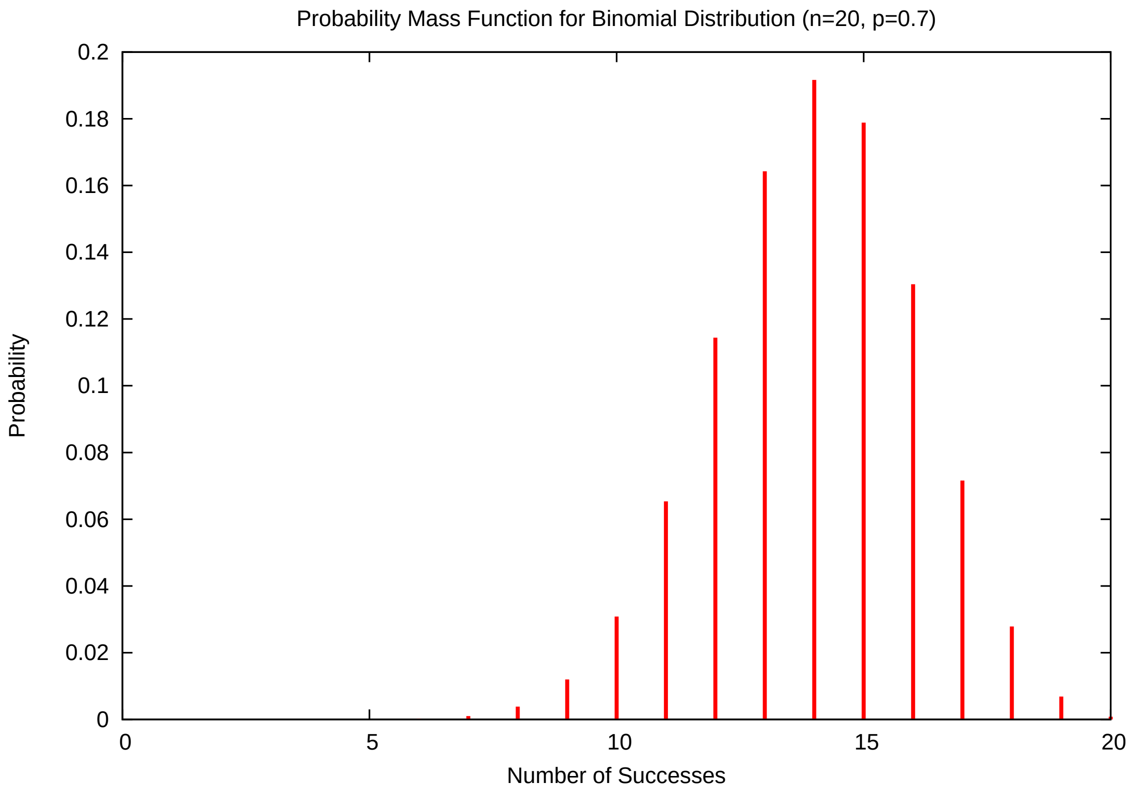


Figure 6.1: Sample probability mass function for 20 Bernoulli Trials with 0.7 probability of success

6.2 Bayesian Statistics and Credible Intervals

Analysis focuses on an array of results of Bernoulli trials, i.e. ones and zeroes, and tries to deduce the probability of another Bernoulli trial resulting in one. To do this, there are two main clues to be used. First one is *evidence*, which is a subset of all possible results, discussed earlier. The second clue is called a prior, which is an estimation of what the probability is likely to be, via a belief, or information gained by observing earlier behaviour of the model.

Let y be the number of ones generated by n Bernoulli trials, and θ be the probability of any trial resulting in one. Note that these n trials are only known experiments, and there are likely to be more experiments generating possibly different results. The question we direct to the data we have is the following: If there are y ones in n Bernoulli trials conducted, what is the probability of next experiment resulting in one, i.e. what is θ in the light of our current knowledge? This can be modelled as the following formulae, based on Bayes' theorem.

$$p(\theta|y) = \frac{p(y|\theta) \times p(\theta)}{p(y)}$$

Here, $p(\theta|y)$ is the posterior probability distribution, which gives us the probability distribution for the next Bernoulli Trial. $p(y|\theta)$ is the evidence (also referred as likelihood), which is the probability distribution given by evident results and $p(\theta)$ is the prior knowledge regarding θ. As $p(y)$ acts as a normalizing constant, this equality can be translated into the following proportionality, as it is commonly denoted.

$$p(\theta|y) \propto p(y|\theta) \times p(\theta); \text{ informally } Posterior \propto Evidence \times Prior$$

It was shown that the evidence is in binomial model. We chose beta distribution to demonstrate prior knowledge, as it suits for probability distribution, integrating to 1. In addition, beta distribution is a conjugate prior distribution to binomial distribution [21], which means that applying beta distribution as prior to a binomial model, posterior is the same family as the prior (beta). This simplifies calculations. As a result, the posterior takes the following form:

$$
\begin{aligned}
p(\theta|Y) &\propto B(y,n) \times Beta(\alpha,\beta) \\
&\propto (\theta^{y} \times (1-\theta)^{n-1}) \times (\theta^{\alpha-1} \times (1-\theta)^{\beta-1}) \\
&\propto \theta^{(y+\alpha)-1} \times (1-\theta)^{(n+\beta)-1} \\
&\propto Beta(y+\alpha, n+\beta-y)
\end{aligned}
$$

In summary, applying a beta prior with hyper-parameters α and β to a binomial likelihood with parameters y, n and θ results in a beta posterior with hyper-parameters $(y+\alpha)$ and $(n+\beta-y)$.

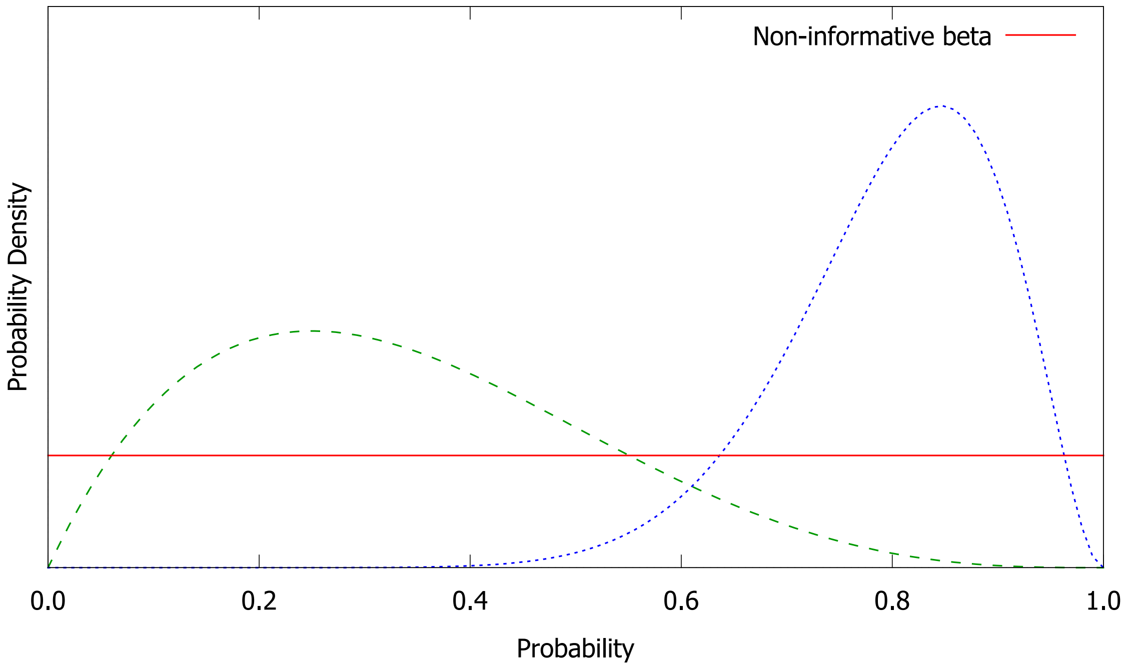


Figure 6.2: 3 Sample beta distributions

After getting the posterior distribution, to finalize the estimation process, we do interval estimation calculations to come up with a credible interval [12]. A credible interval is an interval within the bayesian posterior distribution that covers the actual probability, referenced earlier as θ, with a confidence level γ. In other words, if the probability that θ is within an interval is 95%, that interval is called a credible interval with a confidence level 95%. Note that the confidence level is a parameter itself, and can be adjusted based on the context. Mathematically, for a distribution F of a variable X, a credible interval with a confidence level γ is an interval (a, b) that satisfies the following criteria.

$$Pr(a \leq X \leq b) = \gamma$$

For a given probability distribution, there are more than one credible intervals with the same confidence level. One possibility is to choose the mean of the distribution as the central point of the interval. Another possibility is to find the narrowest interval that gives the specified confidence. Our focus is to find an equal-tail interval, which is the interval for which the probability that θ is below the minimum point is the same as it is above the maximum point.

To calculate a credible interval from a beta posterior, quantile functions are used. A quantile function $Q(p)$, also called inverse cumulative function, is a function that returns value v such that the probability that the variable X is less than or equal to v is p. Mathematically,

$$Q(p) = v \,|\, (Pr(X \leq q) = p)$$

So, the equal-tail credible interval with confidence γ for a beta function with hyper-parameters α and β is given as the following.

$$[qbeta(\frac{\gamma}{2}, \alpha, \beta),\ qbeta(1 - \frac{\gamma}{2}, \alpha, \beta)]$$

where $qbeta$ is the quantile function for beta distribution.

6.3 Performance Analysis & Prediction

Sınavo provides a system to analyse user performances and deduce states of users on different subjects. These analyses are not mere statistical figures, but rather data-driven decisions based on bayesian statistics and credible intervals mentioned earlier.

From a user's perspective, each question the user encounters and tries to solve is a Bernoulli Trial with two possible results: solving it correctly - success, failing to solve it - failure. These statistics are gathered incrementally whenever users solve questions, ready to be fetched.

A user's performance on a subject is modelled as their probability of solving a question in that subject correctly. As this probability is unknown, we apply bayesian statistics to calculate a credible interval and use it as an empirical estimation of the performance of user.

Let us define a user whose statistics will be investigated as an example. Consider a user that just started using Sınavo. Let n_1, the number of questions that user solved on an arbitrary subject S, be 10. Let y_1 be 9, which is the number of questions from those n_1 questions that the user solved correctly. As a result, the user conducted n_1 Bernoulli trials, y_1 of which were successful. Let θ be the probability of that user solving an arbitrary question from that subject correctly, which is deemed to be the performance of that user on that subject. A posterior can be calculated as the following.

$$\begin{aligned}
p(\theta|Y) &\propto B(y_1, n_1) \times Beta(\alpha, \beta) \\
&\propto B(9, 10) \times Beta(1, 1) \\
&\propto Beta(9 + 1, 10 + 1 - 9) \\
&\propto Beta(10, 2)
\end{aligned}$$

As the user is a fresh one without any prior evidence, a non-informative prior is applied, which does not affect the evidence much. For a Beta model, hyper-parameters of a non-informative distribution is $\alpha = 1$, $\beta = 1$, as given in Figure 6.2. An initial informal estimation would be that the user's performance is quite high on that subject, solving 9 questions correctly out of 10. The posterior calculated above generates the distribution given in Figure 6.3, with α_2 being 10, and β_2 being 2.

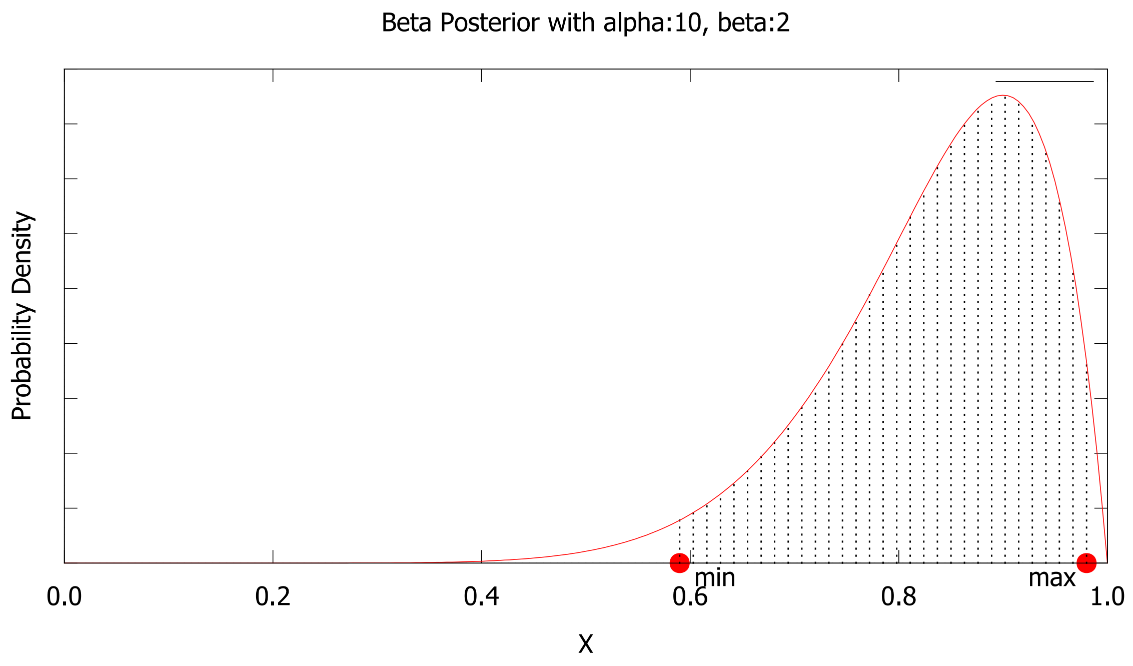


Figure 6.3: Posterior Beta(10, 2) and credible interval with confidence level 95%

As it can be seen, with a non-informative prior and a list of Bernoulli trials with a pretty high success rate, distribution has its mean close to one, which yields that the probability of success is also close to one. The credible interval with 95% confidence, acquired from this posterior is approximately $[0.59, 0.98]$, calculated as the following.

$$min = qbeta(0.025, 10, 2) \approx 0.59, max = qbeta(0.975, 10, 2) \approx 0.98$$

Let us assume that after solving n_1 questions, user solved n_2 questions in the next month, y_2 of which were successful. As users performances change by time, we consider months as epochs and use previous month's posterior, as the prior for the new month. Let n_2 be 20, and y_2 be 5, meaning that user was not as successful at solving questions on subject S this month as the previous month. Using previous month's posterior as the new prior, we get the up-to-date posterior as the following.

$$\begin{aligned}
p(\theta|Y) &\propto B(y_2, n_2) \times Beta(\alpha_2, \beta_2) \\
&\propto B(5, 20) \times Beta(10, 2) \\
&\propto Beta(5 + 10, 20 + 2 - 5) \\
&\propto Beta(15, 17) \\
ConfidenceInterval &\approx [0.30, 0.64]
\end{aligned}$$

Although the user's performance was quite low in the second month, high rate of success in the first month is indicative that it is not the expected performance of the user. It might be the case that some exceptional things happened that lead in low success rate. Applying the prior from the first month adds this possibility, resulting in a mediocre performance expectancy. If the prior was not as good as the first month's performance, calculated posterior would be worse. Assuming there were no priors for the second month's performance, the posterior would be $Beta(6, 16)$. This case, along with the actual prior and actual posterior, is given in Figure 6.4.

It is worth noting that these credible intervals calculated from the posteriors are relatively wide. Stating that a user's performance is probably between 59% and 98% might not be very informative, as a user with 98% performance is a top rated student, but 59% performance might indicate a mediocre success. There are two reasons as to why these intervals are so wide. Firstly, these credible intervals are calculated with a confidence level of 95%, which yields that the actual performance is almost surely within these intervals. And to be so sure, intervals widen to cover most of the probability. Lowering the confidence level

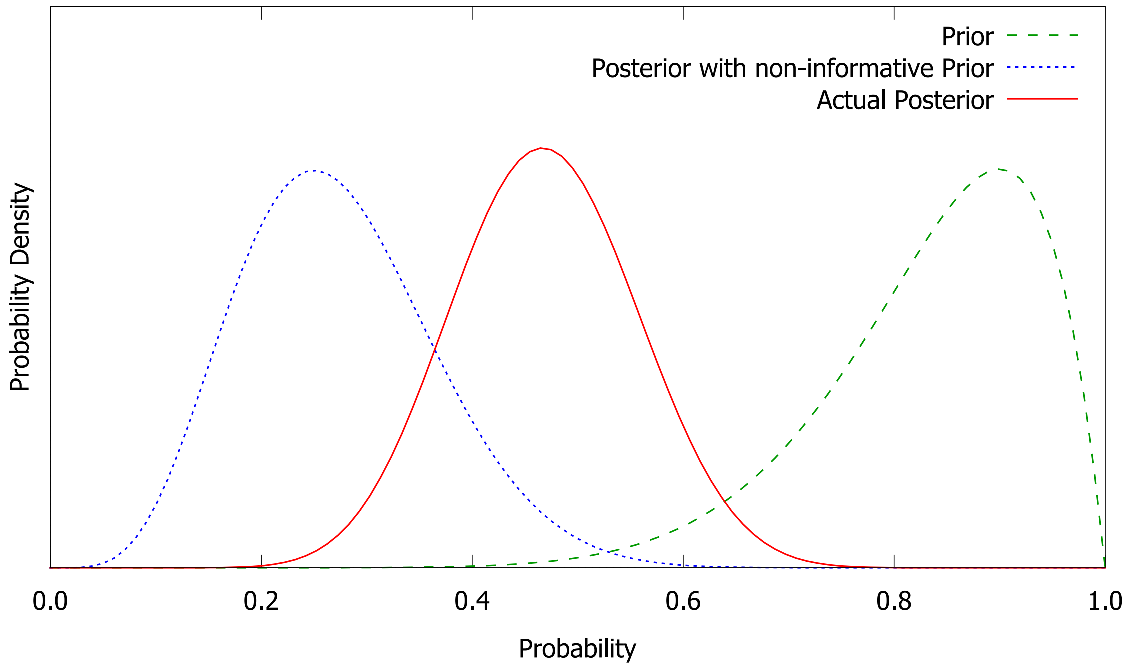


Figure 6.4: Prior & posteriors with actual prior and non-informative prior

would shrink the interval to an acceptable level. For example, instead of requiring a confidence level of 95% and getting interval [0.59, 0.98], choosing a confidence level of 60% results in a narrower interval, $[qbeta(0.2, 10, 2), qbeta(0.8, 10, 2)] \approx [0.75, 0.92]$ (Figure 6.6).

However, lowering the confidence level yields a greater possibility of misleading results. For precision, it is an unwanted property and should be avoided. The second way of narrowing confidence intervals is much more acceptable, which is increasing the data size. In the previous example, the user was assumed to solve 10 questions with 9 correct answers, which generated interval [0.59, 0.98]. Assuming the success rate of the user remains the same, analysing the user's performance over 100 questions with 90 correct answers would generate a narrower interval, $[qbeta(0.025, 91, 11), qbeta(0.975, 91, 11)] \approx [0.83, 0.94]$ (Figure 6.6). Note that the prediction still holds 95% confidence level, but yields a much narrower interval, which is much more informative than the previous one. Shrinking of the interval with wider dataset is only natural, since increase in evident data leads to a more precise deduction.

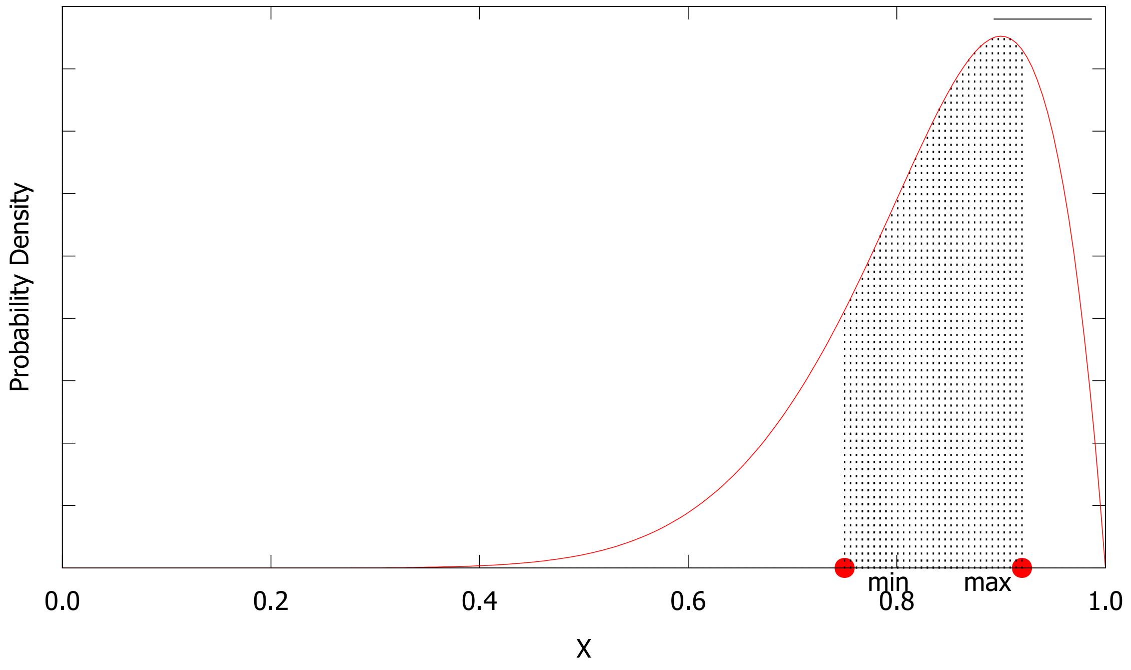


Figure 6.5: Credible interval with 60% confidence level

6.4 Question Difficulty Deduction

Traditionally, a question's difficulty is decided by its creator or a group of experts, based on their intuitions. Although these people have sound understanding of how questions should be classified regarding their difficulties, it is a subjective matter. A question might be regarded as easy by some students, but found to be hard by others. Intuitions might not be enough to come up with an accurate classification. Instead, Sınavo offers a data-driven process to discover question difficulties.

From a question's perspective, it being solved by any arbitrary student is a Bernoulli Trial. If a student solves the question successfully, result of the trial is 1, and 0 otherwise. We model the difficulty of a question as the probability of it being solved correctly by an arbitrary user. Just like user performances, we apply bayesian statistics and credible interval analysis on these bernoulli trials to generate a credible interval for this probability.

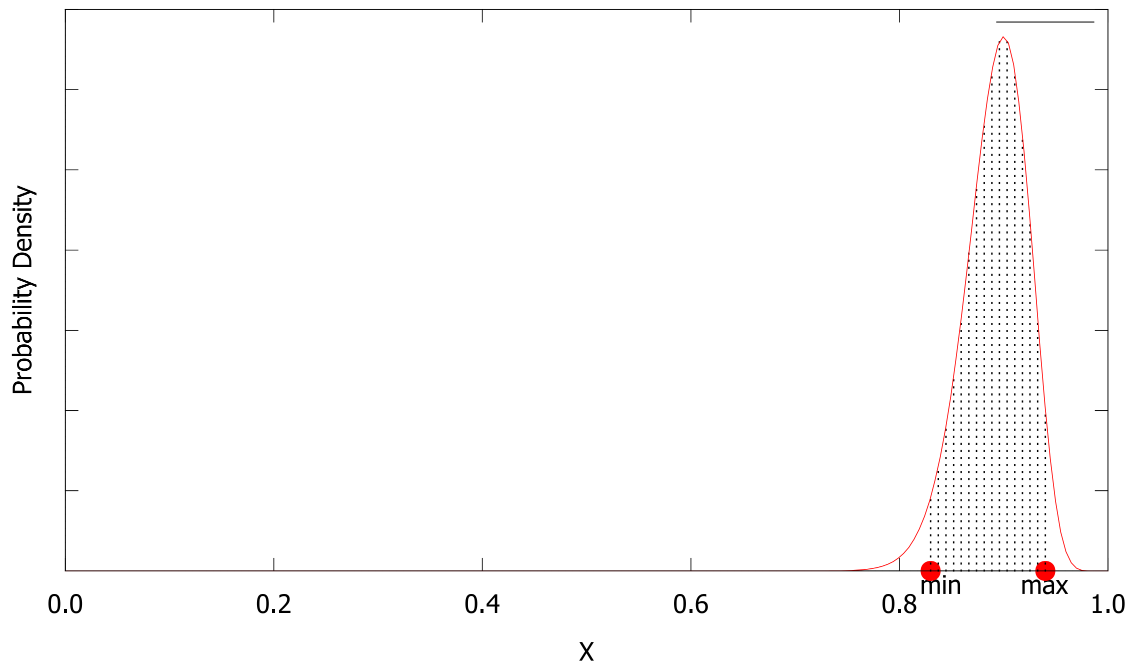


Figure 6.6: Credible interval with more data and 95% confidence level

As we regard difficulties of questions static, we do not apply monthly calculations, using previous month's distribution as a prior. Instead, opinions of experts are used as priors when applicable. When no such data is available, non-informative prior is used. For example, let n, the number of times a question is solved, be 50, and let y the number of times users succeeded in solving that question be 30. Assuming that there is no prior knowledge available, the posterior probability distribution for this question's difficulty (ϕ) will be calculated like the following.

$$\begin{aligned} p(\phi|Y) &\propto B(y,n) \times Beta(\alpha,\beta) \\ &\propto B(30,50) \times Beta(1,1) \\ &\propto Beta(30+1, 50+1-30) \\ &\propto Beta(31,21) \\ ConfidenceInterval &\approx [0.46, 0.72] \end{aligned}$$

Difficulty of a question is usually a label, from a domain of labels such as easy, moderate, hard, instead of numerical values. However, difficulty values of questions actually differ among themselves. Not all questions labelled as the same are at the same level of difficulty. Instead, these labels group questions with similar difficulties; hence, represent intervals. To be able to group the intervals generated by Sınavo system, we need a data-driven method to come up with an *interval* $\rightarrow$ *label* mapping. To do this, we analyse how these intervals are scattered.

First, means of posterior distributions are calculated. Mean for a beta distribution for $0 \leq x \leq 1$ is calculated as $\frac{\alpha}{\alpha+\beta}$. Then, these means are considered as single values and analysed as a new distribution D. Mean μ' and standard deviation σ' of this new distribution are calculated. Using the μ' and σ', difficulty intervals are generated by the following method. As the new distribution is expected to follow a normal distribution, items that are more than three standard deviations farther than the mean are considered as outliers. Hence, items are expected to be scattered over six standard deviation long interval. Splitting this space into the number of labels required, 5 in Sınavo, we get the intervals for difficulties.

A sample labelling procedure is plotted in Figure 6.7. After calculating σ' and μ', difficulty boundaries are calculated. The lowest boundary L_1 is $3 \times \sigma$ units to the left of the mean, and each label interval covers $1.2 \times \sigma$ space as we split $6 \times \sigma$ long space into 5 buckets. Each consecutive boundary forms a difficulty label interval. Considering the fact that x-axis represents the probability of a question being solved, the buckets to the left represent harder questions; whereas, buckets to the right represent easier ones. As a result, following difficulty intervals are calculated:

$[0, L_2]$	very hard
$[L_2, L_3]$	hard
$[L_3, L_4]$	moderate
$[L_4, L_5]$	easy
$[L_5, 1]$	very easy

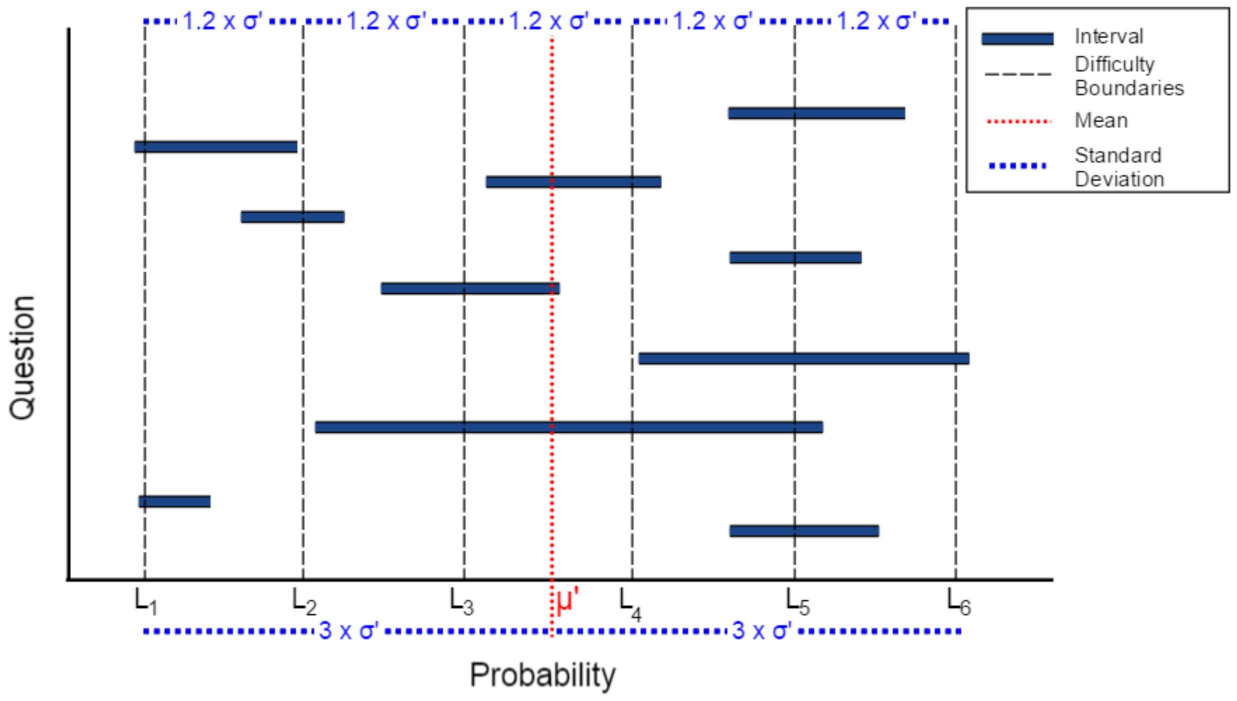


Figure 6.7: Difficulty labels for question intervals

Note that the credible intervals of probabilities for questions being solved are simply stating the actual probability lies within that interval, and nothing more. Hence, we can not just select a label for a question whose mean lies within that label's interval. Instead, each question difficulty interval is compared to all label intervals to see if two intervals have an intersection. If that is the case, it is possible, but not certain, that the question is from that difficulty. As a result, question difficulties are sets of difficulty labels in Sınavo. As discussed earlier, the more a question is solved, the narrower the credible interval for that question will be. That is why all difficulties are expected to converge into having only one label eventually.

6.5 Summary

To summarize, we have introduced two novel ways of using statistics of Sınavo system to make valuable data-driven decisions. In section 6.3, we have shown that by considering acts of solving question as Bernoulli trials, bayesian statistics can be applied to come up with credible intervals. We model performance of a user as the expected probability of that user solving questions correctly. We use stored statistics as the evidence, which is a binomial distribution. We use a non-informative beta distribution with hyper parameters ($\alpha = 1, \beta = 1$) in the beginning, which results in a posterior with beta distribution, as beta distribution is conjugate on binomial distribution. Then, we use these posterior distributions as priors and update credible intervals monthly, to mimic the fact that student performances are dynamic and change over time. Moreover, we have shown that with limited data and high confidence levels, credible intervals are wide and uninformative (see Figure 6.3). However, lowering confidence level (see Figure 6.5), or more preferably, increasing evident data (see Figure 6.6) result in narrower intervals. Consequently, as students solve more questions, estimations will become more precise.

In addition, we apply similar approach to questions to come up with data-driven difficulty estimation, described in section 6.4. We model a question's difficulty as the probability it being solved correctly by any student and try to find that probability. For this, we consider questions being solved as Bernoulli trials, regardless of the student solving them. Applying same approach, we generate credible intervals that correspond to expected difficulties of questions. We also discuss how these intervals can be mapped to difficulty labels by dividing difficulty distributions to intervals for each label. As credible intervals are wide with limited evidence, they can intersect with multiple labels. As a result, initial estimated difficulties can include more than one label. However, as more data is collected, credible intervals shrink and converge to one label.

Chapter 7

Conclusion

7.1 Summary

The world is getting digitalized, and a wide variety of real world data is being collected constantly. In addition, with more and more people getting Internet access, there are a lot of systems with a vast number of users. Such systems are often operate real-time and run into the trouble of storing and retrieving data efficiently. With the needs of software systems changing, database management systems evolved accordingly. For an important amount of time, SQL Servers were enough to handle operations of software systems, but when they started to fail meeting new expectations, NoSQL movement challenged fundamentals of database systems and resulted in a number of new approaches of modelling data.

In this thesis, we have discussed new technologies used to store data on clouds and new means introduced to process this data. We have explained how traditional SQL servers worked and focused on two new database approaches: document based and graph databases. We have exemplified those approaches with widely used database management systems, which are MongoDB for document store and Neo4j for graph database. We have discussed how those approaches modelled data and provide means of storing and accessing that data.

In chapter 4, we have explained Sınavo, which is an online educational system that provides an extensive set of questions, that can be solved in different contexts. Those contexts include competitive games and tests, which are all accessible via a web interface. We have discussed main focuses of this system, such as socialization and ability to provide statistical analysis. In the next chapter, we have presented how different database management systems are implemented for this system. We analysed strengths and weaknesses each approach exhibits and provided a comparison of these approaches on different aspects, such as ease of use, maintainability, and efficiency.

Storing and retrieving Big Data is a problem on its own, but what is being done with this dataset is a fertile field often used incompletely. This is because data gathered for a specific purpose, such as messages from a vastly used social media, also contains possibilities for valuable inferences. In chapter 2, we mentioned that buzz in Twitter is used to estimate how successful a movie will be in theaters. In chapter 6, we present a novel way of using statistics gathered by Sınavo system to estimate students' actual performances. We have acknowledged that success rates of students are subject to change in time and provided a feedback loop that can be used to adjust these estimations accordingly. We have also introduced a way of evaluating question difficulties in a data-driven manner. We have discussed how a bayesian approach can be applied to statistics of questions within Sınavo system to come up with credible intervals. We used those credible intervals to classify questions based on their difficulty levels.

Despite all our efforts, this thesis is naturally not perfect. The drawbacks of this thesis can be summarized as follows.:

- Although we have covered three different types of database management systems, there are a lot more. There are column-oriented databases such as the Apache Cassandra, or key-value stores such as Redis, as well as other instances of the database types we have covered, such as Oracle for relational databases, Titan for graph databases. As discussed before, it is not within the scope of this thesis to compare all existing databases and provide a perfect selection for Big Data projects.

- All the experiments we have conducted were run on a single machine setting. This means that we did not evaluate scalability aspect of these database management systems.
- While evaluating quantitative properties of database management systems, we have focused on read queries. However, write queries might be just as important for projects.

Contributions of this thesis can be summarized as follows:

- In chapter 5, we have discussed design and data model for Sınavo on three database management systems, namely Microsoft SQL Server, MongoDB, Neo4j.
- In section 5.4.1 we have provided a comparison on their non-quantitative attributes and discussed that relational databases are easier maintain, but are less flexible. MongoDB; on the other hand, is easier to use, as there are way less collections than relational databases, as related data is encapsulated within fewer documents. When it comes to designing process, Neo4j is the easiest, considering the fact that software projects are mostly designed in graph-like models, such as entity-relationship diagram.
- In section 5.4.2, we present results of experiments we have conducted on three database management systems. These experiments show that MongoDB performs better on queries that require small number of joins. On the other hand, when the number of joins required increases, such as finding people that a user is connected via multiple friendship relationships, i.e. friends of friends of friends, etc, Neo4j outperform others.
- In chapter 6, we present a novel way to use existing statistics to make data-driven decisions on student performances. In section 6.3, we show how student performances can be estimated. We model student performances as their probability to solve questions correctly. We make use of bayesian statistics and apply beta priors to find credible intervals, which

correspond to student performances. We acknowledge the fact that students' level change in time. We apply monthly updates, using previous month's posterior as the prior for the new month, to mimic this change.

- In section 6.4, we apply bayesian approach on question statistics to estimate question difficulties. We model a question's difficulty as the probability it being solved correctly by any student. We use existing statistics to come up with credible intervals for this probability, and show a way of mapping these credible intervals to different difficulty labels.

7.2 Future Work

As discussed earlier, there are a vast number of new approaches for storing and processing Big Data. Although we have discussed database management systems which we deem important actors, we acknowledge that there are many others, possibly more suitable for different kinds of software systems. An extensive evaluation of those systems and models may be very beneficial to further pinpoint how database selections should be done based on the requirements of projects.

For projects that require an intersection of features different database models offer, there may be systems making use of more than one database management systems. If it is possible to separate the data into mutually exclusive subsets, such a configuration might work well. However, this is usually not the case and data is connected overall. In such configurations, if more than one DBMS's are used, a new problem of keeping those DBMS's in sync emerges. In addition, new technologies can be designed as hybrids of existing technologies, such as a graph database, vertices of which are documents.

We have presented a novel way of estimating student performances and question difficulties based on statistics of Sınavo system. For now, these estimations are used solely to inform students and teachers. However, changes in performances may be analysed to do a lot more. For example, an intelligent system may be developed to match decreases/increases in those performances to behaviours of students. It might estimate the optimal intervals in which students should repeat certain subjects to prevent their performances from decreasing. In addition, as students have target scores, an intelligent system can analyse their performances and provide an optimized schedule to achieve those targets. For example, a subset of students may be able to improve their performances on a specific subject easier than others, with possibly diminishing efficiency. Such an intelligent system can make data-driven decisions to estimate optimal levels for students on subjects and encourage an optimized schedule to reach those levels.

BIBLIOGRAPHY

[1] Neo4j, the world's leading graph database. http://neo4j.com/product/, 2015.

[2] Sitaram Asur and Bernardo A Huberman. Predicting the future with social media. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on*, volume 1, pages 492–499. IEEE, 2010.

[3] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1071–1080. ACM, 2011.

[4] N. Bressan, A. James, and C. McGregor. Trends and opportunities for integrated real time neonatal clinical decision support. In *Biomedical and Health Informatics (BHI), 2012 IEEE-EMBS International Conference on*, pages 687–690, Jan 2012.

[5] Kristina Chodorow. *MongoDB: the definitive guide.* " O'Reilly Media, Inc.", 2013.

[6] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[7] Edgar F Codd. Recent investigations in relational data base systems. In *IFIP congress*, volume 74. Amsterdam., 1974.

[8] Edgar Frank Codd. Further normalization of the data base relational model, data base systems, courant computer science symposia series 6, r. rustin, 1972.

[9] Douglas Crockford. *JavaScript: The Good Parts: The Good Parts.* " O'Reilly Media, Inc.", 2008.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[12] Ward Edwards, Harold Lindman, and Leonard J Savage. Bayesian statistical inference for psychological research. *Psychological Review*, 70(3):193, 1963.

[13] Ronald Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Systems (TODS)*, 2(3):262–278, 1977.

[14] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*, 2012.

[15] John F Gantz and David Reinsel. The expanding digital universe: A forecast of worldwide information growth through 2010. IDC, 2007.

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[17] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[18] Colin J Ihrig. Javascript object notation. In *Pro Node. js for Developers*, pages 263–270. Springer, 2013.

[19] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual*

ACM symposium on Theory of computing, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[20] Simon Josefsson. The base16, base32, and base64 data encodings. 2006.

[21] Paul H Kvam and Brani Vidakovic. *Nonparametric statistics with applications to science and engineering*, volume 653. John Wiley & Sons, 2007.

[22] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.

[23] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets.* Cambridge University Press, Cambridge, 2012.

[24] Ronald Rivest. The md5 message-digest algorithm. 1992.

[25] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases.* " O'Reilly Media, Inc.", 2013.

[26] Robert Schlaifer and Howard Raiffa. Applied statistical decision theory. 1961.

[27] Gautam Shroff. *The Intelligent Web: Search, smart algorithms, and big data.* Oxford University Press, 2013.

[28] Jeffrey D Ullman et al. *A first course in database systems.* Pearson Education India, 1982.

[29] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. Data mining with big data. *Knowledge and Data Engineering, IEEE Transactions on*, 26(1):97–107, 2014.