EFFICIENT TRANSFER OF SCIENTIFIC DATA

by

SAMEER GAHERWAR

(Under the Direction of John A Miller)

ABSTRACT

Dataset sizes are growing rapidly, so it is very important to be able to efficiently model and transfer large datasets over the network. In this thesis, we have addressed some of the issues involved by presenting the GlycoVault Data Transfer Module (GDaTM), which is implemented using some of the latest technologies for effectively modeling and transferring large datasets. Transfer of large datasets goes hand in hand with data storage, which is used to store the transferred data. We have conducted a meta analysis comparing different types of database technologies as well as experiments comparing the performance of database insertions and retrievals for two types of data stores. We have also conducted experiments comparing various means of data transfer, including multi-part vs. streaming and with vs. without compression. We have also compared two well-known data serialization and deserialization API's. Lastly, we have analyzed alternative data stores, including Scalable SQL, NoSQL and combinations of both.

INDEX WORDS: JSON, RESTful web services, NoSQL, Graph Isomorphism.

EFFICIENT TRANSFER OF SCIENTIFIC DATA

by

SAMEER GAHERWAR

BE, University of Pune, India, 2010

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2014

EFFICIENT TRANSFER OF SCIENTIFIC DATA

by

SAMEER GAHERWAR

Major Professor:	John A Miller
Committee:	William York
	Krzysztof J. Kochut

Electronic Version Approved:

Julie Coffield
Interim Dean of the Graduate School
The University of Georgia
December 2014

ACKNOWLEDGEMENTS

I would like to thank Dr. Miller for his constant guidance throughout my academic career at UGA and helping me improve my overall technical skills. I would also like to thank Dr. Kochut for his inputs in my research work and also advising me on designing the API. I would like to thank Dr. York for his valuable inputs for modifications in my thesis. I would also like to thank Rene Ranzinger for helping me throughout my role as a Research Assistant.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

Today's ever expanding data poses a great challenge to be able to build a scalable module, which can model and efficiently transfer large amounts of data over a network for storing and retrieving purposes. With the transfer of large data comes the issue of performance; we need to have a reliable, efficient and fast way of sending the data over the network. With the issue of performance and limited network bandwidth the question arises to have an appropriate compression method, which can significantly compress the data without being too time consuming as most of compression algorithms use a significant amount of time and resources. Now with this problem of limited bandwidth we also have to make sure that we do not store redundant information in our database, because having the same data, which is huge, more than once will significantly affect the performance of database related operations. Having a scalable module to send data alone will not solve the problem of efficient data transfer, as we also need to have an appropriate data store, which can handle this data. To summarize, all the above-mentioned issues or concerns are interconnected. Addressing one of them leads us into exploring the other. If we can address most of these issues we can come up with a module, which can efficiently transfer data and moreover can be scalable.

In this thesis work we have implemented the GlycoVault Data Transfer Module (GDaTM), which is a prototype module, to address most of the above-mentioned issues. Additionally, we propose a meta analysis for the appropriate choice of database for data store, which can save such data. In this thesis we will be discussing the following issues: In Chapter 2 we discuss the

background of the terms and concepts, which are crucial in understanding the flow of this thesis. In Chapter 3 we discuss the current and past trends driving us as to why we see a need for more efficient data transfer and what is the reason we pursue addressing this issue in particular. In Chapter 4 we discuss some of the related work and some of the related technologies that have close relationships to the prototype we have developed. In Chapter 5 we discuss how to maintain uniqueness in large data sets, which leads us into exploring the field of Graph Isomorphism, and two algorithms for solving this problem. In Chapter 6 we discuss GDaTM in detail and explain its three vital modules. In Chapter 7 we discuss the implementation of GDaTM. In Chapter 8 we discuss the experiments, which we have carried out for various means of data transfer and also compared database insertions with two techniques. We also present a meta analysis on various types of databases and their usage. In Chapter 9 we present the conclusion as to what we think are probable solutions for transferring large amounts of data with an appropriate choice of data store.

CHAPTER 2

BACKGROUND

The problem of transferring large amounts of data has been a significant challenge in the present and the past as well. In the early 90's this problem was even more challenging due to limited availability and high operational costs. In the past a gigabyte was considered as huge data. Now with the changing times the problem is being amplified. Now the operational costs have significantly gone down, network bandwidth have significantly improved but the problem we face now is how to efficiently use the network bandwidth and other resources when we have scaled up to handling huge data sets in the terabyte or petabyte range [33]. Some of the scientists have predicted by 2015 world will see a Zettabyte (1,000 000,000,000,000,000,000 bytes) of data. In this era of such exploding data some scientists have raised concerns for some fields of study where data to be analyzed and processed is still shipped in storage devices instead of network due to its enormous size. Let us discuss the basics of some of the terminologies, which we are going to use later in this thesis.

2.1 HyperText Transfer Protocol (HTTP)

HTTP (HyperText Transfer Protocol) is a protocol used by World Wide Web. It is responsible for defining how messages are formatted and transmitted, and what actions Web servers and browsers should take for a particular service. HTTP is termed as a stateless protocol because it executes each request independently, without preserving any knowledge of the requests that were served before [10, 19].

2.2 Representational State Transfer (REST)

REST (Representational State Transfer) is an architectural model for designing web applications. It uses HTTP as an underlying protocol to transfer messages over the network. It relies on a stateless, client-server, cacheable communication protocol. As opposed to their complex counterpart mechanisms such as RPC (Remote Procedure Calls) or SOAP (Simple Object Access protocol) to connect between machines, REST makes simple HTTP calls between machines. RESTful applications use HTTP POST requests to post data (create), PUT to update data, GET to read data and DELETE to delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations. Despite being simple, REST is fully featured; and provides seamless data exchange through RESTful web services. RESTful web services interchange data using JSON (JavaScript Object Notation) or XML (EXtensible Markup Language). We will discuss JSON in the next sub section in a little more detail as it is proved to be more efficient data interchange format than XML [2,20].

2.2 JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format.

"JSON is built using two popular structures:

- An unordered collection of name/value pairs. It can be visualized as an object or keyed list.
- An ordered list of values. In most languages, this can be seen as an array, vector, list, or sequence.

A JSON object is an unordered set of name/value pairs. An object begins with {(left brace) and ends with} (right brace). Each name is followed by: (colon) and the name/value pairs are

separated by, (comma)" [43]. General grammar for a JSON object can be defined as {"name": value}, where value can be a string or a number.

An array can be defined as an ordered set of name value pairs, which are represented as [object], where this object can be a value, i.e., either a string or a number or a traditional name/value pair, i.e., a single object or an object containing more arrays within [43].

2.3 NoSQL Databases

NoSQL (often interpreted as **Not Only SQL**) derives its name because of its capabilities to provide mechanisms for storage and retrieval of data using ways other than tabular or relational arrangement of data, which is fundamentally different from the relational databases. The data structure differs from the RDBMS, and therefore some operations are faster in NoSQL and some in RDBMS [14]. The main types of NoSQL databases are briefly discussed below:

2.3.1 Key-Value Store

A Key-Value store uses a hash table in which there exists a unique key and a pointer to a specific data. Typically this data store stores the data as a JSON or BLOB (Binary Large Object), which can be represented using a String against a key [24, 25].

Example: Amazon S3 [24, 25].

2.3.2 Document Store

A Document Store, organizes the data in a collection of key value pairs, and may be compressed. A Document Store is very similar to a key-value store, but the only difference is that the values stored (referred to as "documents") have some structure and encoding mostly in the form of JSON, BSON (which is a binary encoding of JSON objects) [24, 25].

Example: MongoDB [24, 25].

2.3.3 Graph Database

A Graph Database uses nodes and edges to represent and store data. These nodes are organized by some relationships with one another, which are represented by edges between the nodes. Both the nodes and the relationships have some defined properties [24, 25].

Example: Neo4j [24, 25].

2.3.4 Column Based Store

A Column-Oriented database, stores data in cells grouped in columns of data rather than as rows of data. As its name suggests, reads and writes are done using columns rather than rows. In comparison to most relational DBMS that store data in rows, the advantage of storing data in columns is fast search, access and data aggregation [24, 25].

Example- HBase, Cassandra. [24, 25].

2.4 GlycoVault

"The primary goal of GlycoVault is to provide infrastructure for research in bioinformatics. GlycoVault is designed not only to store data but also to visualize and analyze data. GlycoVault provides a means of storing and retrieving data to support glycomics research at the Complex Carbohydrates Research Center (CCRC) at the University of Georgia. These data include quantitative Real-Time Polymerase Chain Reaction (qRT-PCR) data as well as basic glycomics data. GlycoVault not only provides scientists with a robust means of retrieving and analyzing their results, it also provides an online store of the knowledge and data collected by the CCRC. GlycoVault provides access to data and knowledge stored in form of Relational Tables, Object Model and Spreadsheets" [50]. GlycoVault's service layer, which hosts web services facilitates the development of methods for querying the knowledge and exporting the results in formats (such as JSON) through the GlycoVault Data Transfer Module. Figure 1 shows a high level

architecture for GlycoVault. As we can see in Figure 1, all the workflows for example (IDAWG, qRT-PCR, Simian Tools) submit to and retrieve data from GlycoVault through GDaTM. GDaTM can be visualized as a API which is embedded in a workflow and helps the workflow to query the knowledge in GlycoVault and facilitate the retrieval of the data and also helps in entering new data.

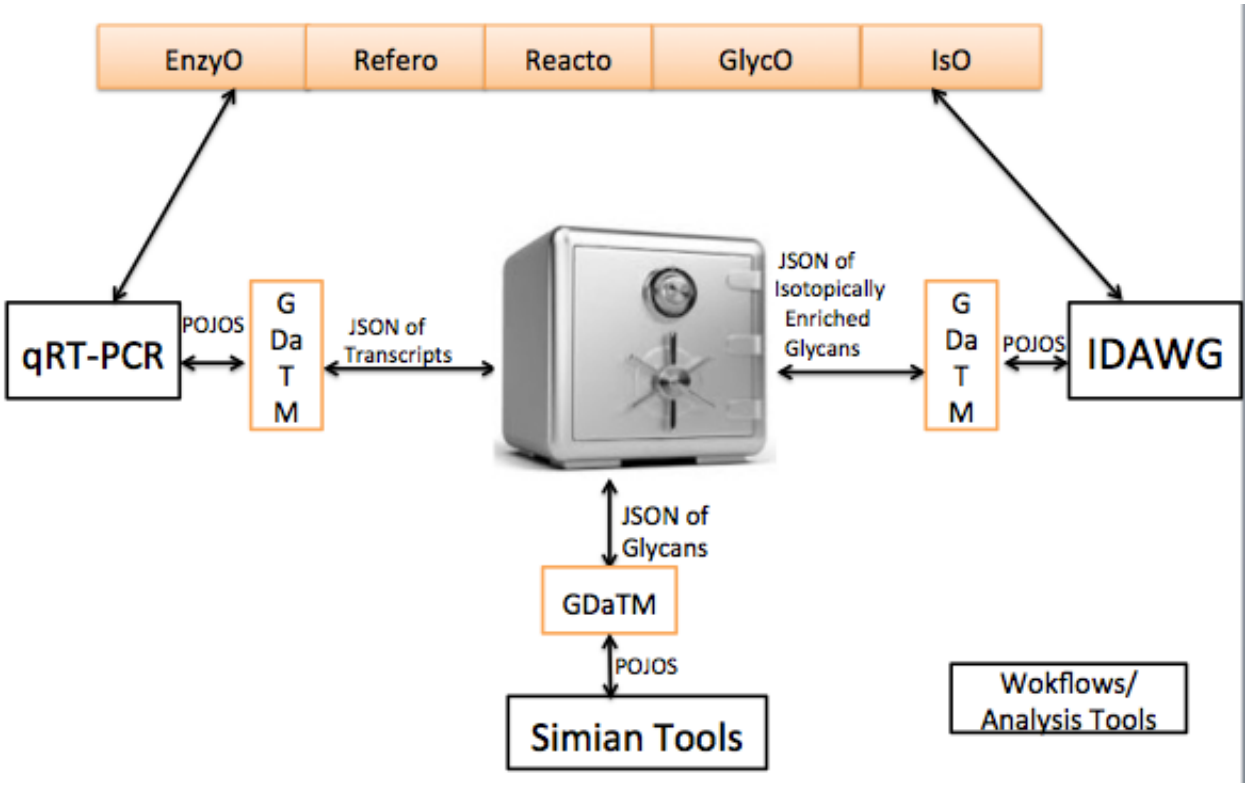

Figure 1: Architecture of GlycoVault

Let us discuss briefly Figure 2, which represents the UML class diagram for GlycoVault. If we look at the Table 1, we can see some of the important classes, which will be the focus of this thesis. Some of the important packages in GlycoVault are also mentioned below in Table 2.

Class Name	Description
BiologicalSample	This class is parent class for SourceSample and DerivedSample on which an Experiment is performed
Composite	Composite can hold any type of Value i.e. a Vector, Scalar or a Composite
DerivedSample	A Derived Sample is usually based or derieved from a SourceSample
DescriptorType	DescriptorType can be of type Group or Simple type
Experiment	Generic class Experiment that can represent several types of Experiment conducted in CCRC.
ExperimentDesign	ExperimentDesign required for an Experiment
ExperimentStep	ExperimentStep(s) are used for constructing Experiment Design
GroupDescriptor	GroupDescriptor can hold several GroupType(s)
GroupType	One of the type of GroupType and also it can hold several SimpleType(s)
GVValue	GVValue is a parent class for the values an experiment can produce
GVVector	This class models GVVector which extends GVValue
LiteralDescriptor	LiteralDescriptor which is of type SimpleDescriptor
MolecularObject	MolecularObject which can be a Gene or a Glycan
OntoDescriptor	OntoDescriptor is one of the type of SimpleDescriptor
Observable	Observable is the name for the output entities in the experimental data
Parameter	Parameter can be defined as a conditional entity that may be required for a protocol for example temparature
ParameterValue	ParameterValue is a value associated with a particular parameter
PhysicalObjectType	PhysicalObjectType which can hold several descriptor type
Protocol	Protocol are nothing but name of the experiments steps which are associated for an Experiment Step.
ProtocolDesign	ProtocolDesign on which a Protocol can be based upon.
ScalarValue	ScalarValue which extends GVValue
SimpleType	One of the DescriptorType(s)
Task	Task are series of steps needed to complete an Experiment
SourceSample	Experiments are conducted on SourceSample which extends BiologicalSample

Table 1 List of few important classes in GlycoVault

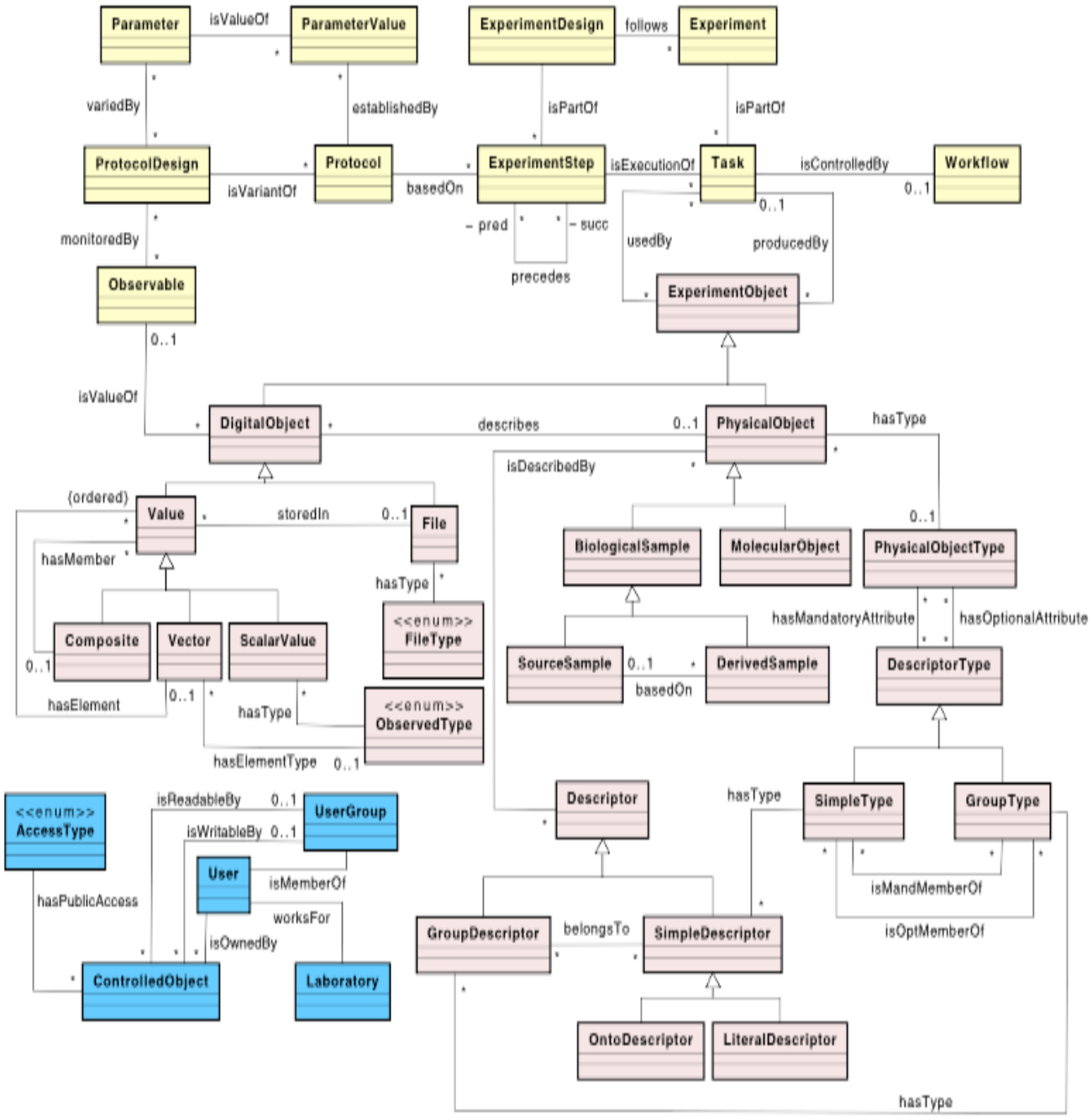

Figure 2: UML for GlycoVault (Courtesy GlycoVault Team)

Package Name	Description
manage.generated	This package contains all the generated classes generated by persistence module
object.association	This package contains interface for all the relationships between classes
object.association.impl	This package contains classes which the implement the interface of the relationships
object.entity	This package contains classes which implements the interface for the classes discusses in Table 1.
object.entity.impl	This package contains classes which implements the interface for the classes discusses in Table 1.
service.generated	This package contains the interface for auto generated services
service.generated.impl	This package contains the classes which implement the interface for auto generated services

Table 2: Package names in GlycoVault

CHAPTER 3

MOTIVATION AND OBJECTIVES

Over the past five years, the emergence of huge data sets and data-intensive science are fundamentally altering the way researchers work and their ability to move their data in every scientific discipline. “Biologists, chemists, physicists, astronomers, earth and social scientists are all benefitting from access to the tools and technologies that will integrate this "big data" into standard scientific methods and processes” [16]. There can be many interpretations for “big data”, which differ depending on the field of study for, e.g., computer science, financial analysis, or entrepreneurship. Regardless of the field of study, they all have one thing in common, i.e., there is a significant growth in the ability to capture, aggregate, and process an ever-greater volume, velocity, and variety of data. Data are now available faster, have greater coverage and scope, and include new types of observations and measurements that previously were not available [16, 27]. Nowadays, there is a concept of “Internet of Things” which is a term used to describe the ability of various devices to communicate with each other using embedded sensors. These devices can be in various locations but they have one thing in common, i.e., they all transmit, compile and analyze data over the Internet. Nowadays, researchers are capable of collecting vast quantities of data through computer simulations, low-cost sensor networks and highly instrumented experiments, creating a huge data flow. As the data sizes are growing constantly, a significant amount of resources are required to process and analyze them [33]. As the computing for these data requires sophisticated hardware, which is not cheap, these researchers are moving toward the idea of moving there data to the cloud or to transfer their data

to a cluster with the high computing power which is usually shared among many researchers. Instead of setting up clusters or supercomputers of their own, which can be costly, many scientist are turning to a cheaper way to run their experiments on a cluster via renting or sending their data and executing them on a remote cluster. As described above many scientific disciplines have become data-driven [16]. For example, a modern telescope has a very large digital camera. The Large Synoptic Survey Telescope (LSST) scans the sky, recording 30 trillion bytes of image data every day. The Large Hadron Collider (LHC), a particle accelerator that studies the Universe generates an estimated 60 terabytes of data per day – 15 petabytes (15 million gigabytes) annually [22]. "Glycomic Elucidation and Annotation Tool, GELATO, is a semi-automated MS/MS annotation tool that rapidly matches hundreds of experimental MS/MS spectra with theoretical glycan fragments from highly curated default glycan databases known as SweetyN and SweetyO" [42]. GELATO, software made by CCRC, uses large data sets, which are analyzed by this software. Many scientific projects are proposed and are underway in a wide variety of other disciplines, ranging from biology to environmental science to oceanography or bioinformatics. All these projects have one thing in common, i.e., they generate large quantities of data and in some cases with a high velocity. Moreover, it becomes infeasible to replicate copies in house for individual research groups, so there is a need to construct a large data center that can run the analysis on these huge data for all of the registered scientists. This will require moving such big data sets, which can be a costly and cumbersome without the right technology. Even having all the technology, does not make it a trivial operation. There are several reasons we might have to move these big data sets, one of them could be if we decide to move our data to a cluster or to a central repository, which has large capacity in terms of storage and computing power, and also provide a wide range of access. Now to aggregate this large volume of data to a

central cluster or a repository it would be fair to assume that most of this transfer would be done through the means of a network. This problem not only raises the question of how efficiently we can access the data, but also how effectively we can model these data. Also, we would require a robust mechanism in form of a program, which can move these large volumes of data and an effective data store, which can consume this large volume of data. Now, having discussed why data transfer is needed, we would also require to have an understanding as to which type of database to choose. It should be scalable and can handle this huge data flow. Regarding database scalability, we need to understand how an RDBMS will behave in this scenario where we have to deal with large volumes of data. We need to comprehend the behavior of the RDBMS transactions, which will eventually deal with large volume of data. It becomes necessary to have understanding of a transaction, what guarantees it can provide and how it will behave in an environment, which can be challenging for many databases. Understanding of these problems can finally make us understand which of these guarantees are provided by an RDBMS and which of these are not provided by NoSQL databases. Specifically, *atomicity*, *consistency*, and *durability* are important for the purposes of this answer. *Isolation* may be relaxed depending on the application [21]. Keeping the above database problems in mind we look for some commonalities between an RDBMS and a NoSQL data store. We look for a middle ground where we can use the good properties of both kinds of database and build a system, which can support scalability and choose an appropriate database or appropriate combinations of databases.

CHAPTER 4

RELATED WORK

In this section, we discuss some of the earlier and current research that is being carried out in the field of transferring large amounts of data, how this data is being modeled and what are some of the popular databases which help in storing this huge data and thus completing data transfer. As stated in "Amazon S3 for Science Grids: a Viable Solution?" which discusses the efficient approach for transferring and computing of big scientific data. This system also uses REST, SOAP and BitTorrent services. In this paper they have predominantly used RESTful web services for large data transfer [1]. Another popular lightweight REST API for RDF data is NanoSparqlServer [8]. According to a study conducted by Intel in "Big Data Technologies for Ultra-High-Speed Data Transfer in Life Sciences" they have discussed an example of genomics data, which is huge, and the need to transmit terabytes of genomic information between the websites worldwide is both essential and daunting at the same time [7]. As discussed by DDN (Data Direct NETWORKS), a well-known company, which provides scalable storage infrastructure for big data and cloud applications, most of the data that is generated today is unstructured information in the form of images, video, sensor data, etc. According to DDN unstructured data is mostly stored as an object storage (architectural style where data is organized in form of objects rather than files). As we have mentioned how today's trend is to move large amounts of data to cloud for storage or to transfer it on high performance clusters on remote locations. DDN has implemented, "WOS", is a high-performance object storage platform designed to easily store petabytes of unstructured data, which can provide good availability with

its high-performance REST API [54]. Netflix, one of the giants in providing online streaming of videos and movies also uses REST API for its data transfer and streaming needs [38]. Google's BigQuery, is one of the popular applications used for querying massive datasets that can be cumbersome and expensive without the right infrastructure. Google BigQuery provides a REST API to transfer these huge data sets to Google's cloud servers [28]. Now that we have discussed some of the popular work related to data transfer using REST API's, let us switch our focus to current and past work done for representation and modeling for these large data. According to the study conducted in "Comparison of JSON and XML Data Interchange Formats: A Case Study" it is shown that JSON is much faster and efficient way of modeling data than XML as it uses fewer resources than its XML counterpart. In today's data transfer needs network, bandwidth is a bottleneck and with the JSON representation, this problem if not completely eradicated, at the very least, can be reduced [5]. Another study explains that the data can be modeled as JSON for big data analytics and how this data can be transferred using web services using APIs which can allows developers to easily integrate diverse content from different web-enabled system, e.g., REST for invoking remote services [4]. Also, JSON was invented as a lightweight alternative to XML but some researches have commented if we look at the current trend, which is heavily biased towards use of JSON it looks as if JSON has challenged the mere existence of XML. As stated in "Seven Challenges for RESTful Transaction Models", transaction processing is one of the essential features of enterprise information systems and choosing the right transaction models for transaction processing in RESTful services is very important. This paper also presents several RESTful transaction models and also suggests that if we need to preserve ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability) properties for a resource, the best way would be to send the resource in one request rather than having it sent in

several requests. This research also states that a system is responsible for its own mechanisms to preserve ACID properties; a REST service will only provide a resource, which can help the system to achieve its transactional properties [39]. In “Scalable SQL and NoSQL Data Stores” the author presents both the pros and cons of Scalable RDBMS and NoSQL and also describes how scalable RDBMS can achieve scalability like NoSQL while preserving all the ACID properties of a transaction [40]. As stated in the study “NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison”: NoSQL databases are getting popular and they are providing a viable alternative for storing big data not only efficiently but also allow the data retrieval to be much faster [9]. Now turning our attention to the methods of compression there are several known compression algorithms. We will discuss two of the most commonly used algorithms, i.e., GZIP and ZIP. In this article, well-known compression techniques have been compared. This article explains some of the critical differences between the two algorithms, i.e., with GZIP, we archive all the files into a single tarball before compression. In ZIP files, the individual files are compressed and then added to the archive. In this article they have stated that GZIP can achieve better compression compared to ZIP [14]. In the study conducted by “A Quick Benchmark: Gzip vs. Bzip2 vs. LZMA”, it is shown that Bzip2 creates almost 15% smaller file size than GZIP but GZIP is faster in compression and decompression; GZIP is around 12 times faster than Bzip2 [3].

CHAPTER 5

GDaTM

5.1 Architecture

The GlycoVault Data Transfer Module (GDaTM) is a module, which can be visualized as a client API that invokes web services on GlycoVault. It is a critical module as this enables various workflows to send and receive data from GlycoVault. As shown in Figure 1, GDaTM is a client API, which can be embedded in a workflow. It accepts input as POJO's (Plain Old Java Objects) from the workflow. These POJO's contain a variety of data, e.g., some of the crucial ones are Experimental data, ExperimentDesign, Sample and ProtocolDesign. These POJO's are then serialized into their respective JSON representations. This JSON is then compressed and sent over to GlycoVault using a RESTful client, which is one of the key components in GDaTM. Various kinds of data transfer currently supported in this version of GDaTM are shown in Table 3. Data transfer has to be in a certain order: the order being all the necessary information for the experimental data, i.e., the sample it refers to and the experiment design it is based on should be present in GlycoVault. GDaTM sends the JSON data using streaming which is compressed using GZIP compression algorithm before sending it. All the services that are invoked by GDaTM have a JSON response. Whenever a service is invoked using GET method a JSON representation of the model is received as response by the client, which invoked the service. This JSON response is consumed and the POJO's are constructed. GDaTM has three major components and one of them holds all the models for the data transfer, some of the key UML class representation

of models are discussed later in this chapter. These UML class models closely resemble the overall GlycoVault UML model shown in the Figure 2.

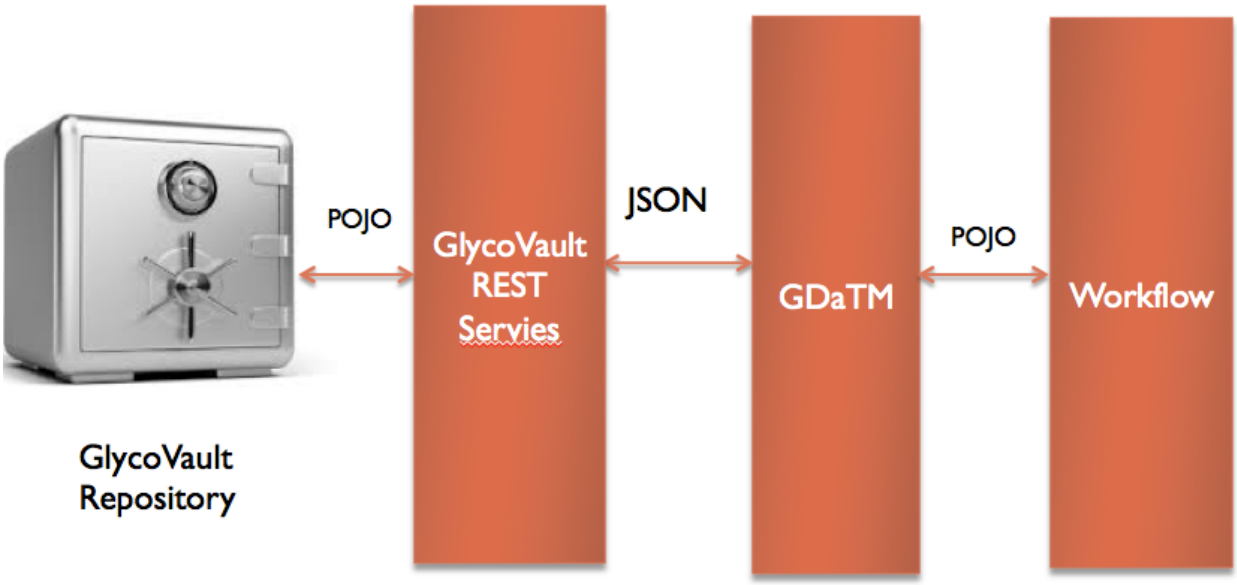

Figure 3: GDaTM architecture

GDaTM invokes services on GlycoVault's service layer. We will see some of the class diagrams of key data representations, which are sent to GlycoVault using RESTful services invoked by GDaTM. These UML models provide a grammar for these objects to be translated into their respective JSON representations.

5.2 SourceSample and PhysicalObjectType UML Model

Let us discuss class diagram in Figure 4, which represents the UML for the SourceSample and PhysicalObjectType representations. It represents the relationship between the SourceSample and PhysicalObjectType. As shown in the Figure 4, the UML representation for SourceSample also shows the relationship each SourceSample has with Descriptors. A SourceSample can contain several descriptors and if we compare this model with Figure 2 we can see how closely it adheres to the interconnectivity of classes in GlycoVault. This close adherence holds true for all the models discussed later in this chapter. Later in Chapter 7 we discuss the model translation

into their respective JSON representations as part of the discussion of the implementation of GDaTM.

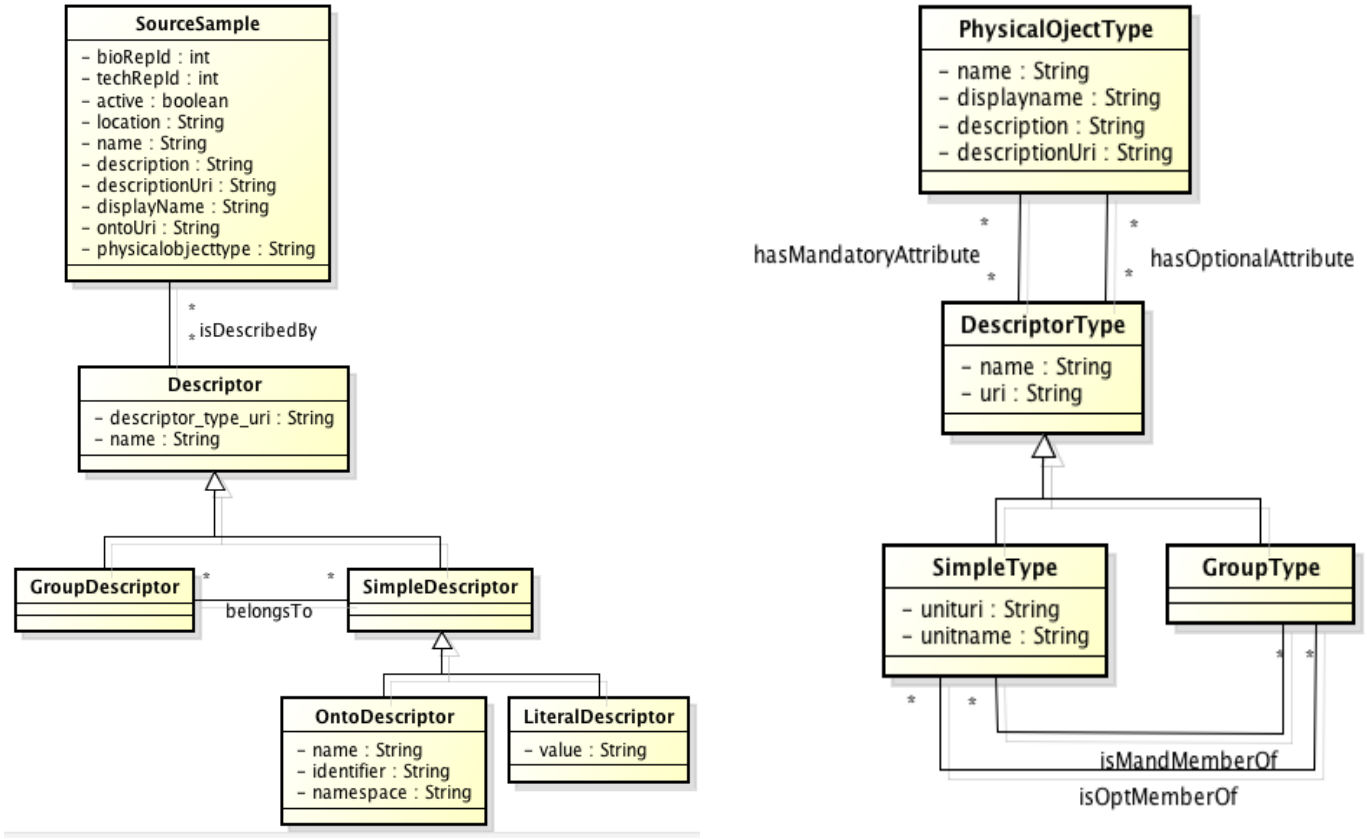

Figure 4 Class diagram to represent Sample and PhysicalObjectType (Courtesy Glycomics Group)

5.3 Experiment UML

Now let us discuss one of the very important UML models, i.e., the one for Experiment. This UML representation, as shown below in Figure 5 shows the classes that model the Experiment data. These classes are in close adherence with the class relationships in the GlycoVault UML. This UML describes an Experiment as having at least one task (usually experiment has several tasks). These individual tasks can have several inputs and a task can generate several outputs. These outputs can be either a BiologicalSample or a MolecularObject. A Task can have list of Value(s), which are generated by the output. For a successful insertion of Experiment in GlycoVault, we need to have the ExperimentDesign and the SourceSample(s), to which that particular Experiment refers to.

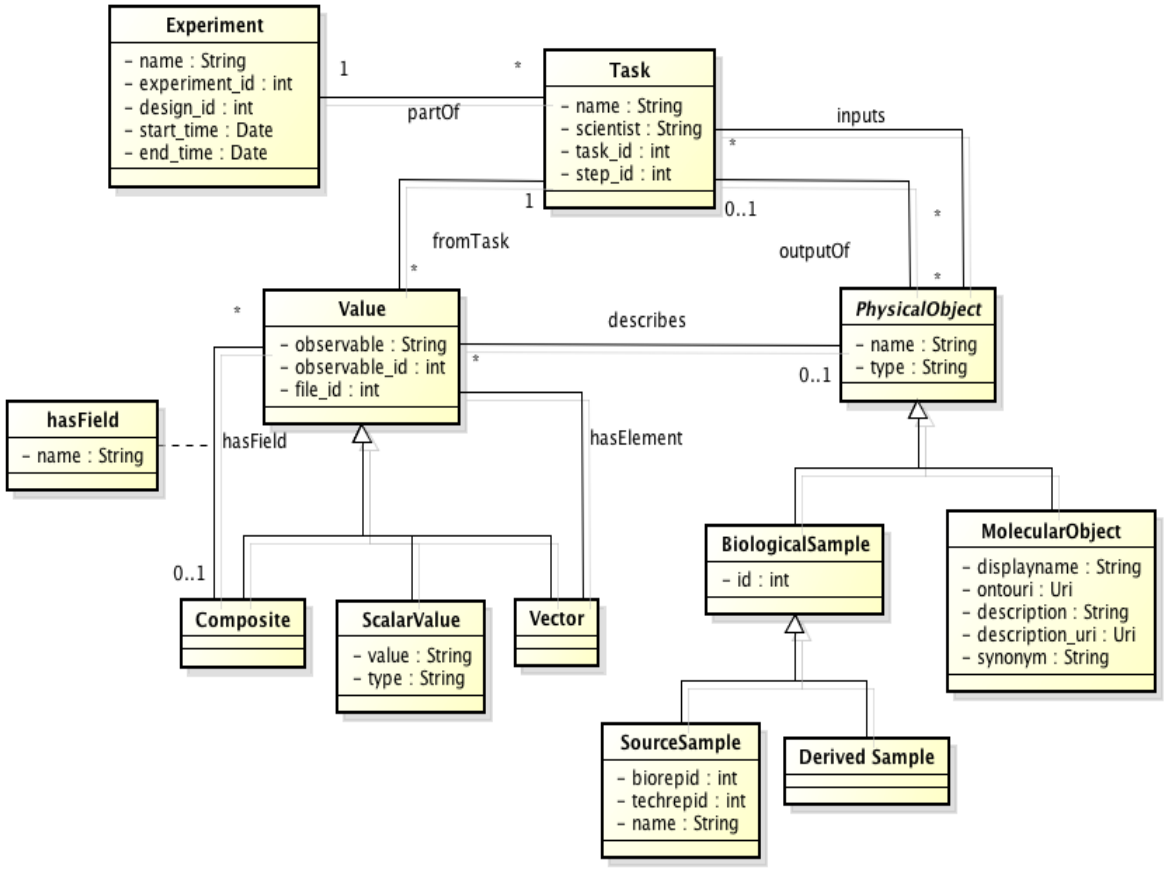

Figure 5: UML Class diagram for Experimental data (Courtesy Glycomics Group)

5.4 ExperimentDesign UML

Now, one of the key data transfers is for the ExperimentDesign, which is shown in Figure 6. These classes are in close adherence with the class relationships in the GlycoVault UML. This UML models some of the key data components such as Protocols, which are variants of ProtocolDesign. List of Observable(s) that are associated with an ExperimentStep(s) forms a complete ExperimentDesign. Typically, Observable(s) and Parameter(s) can be created before and can be referenced at the time of ExperimentDesign submission. The ExperimentDesign UML model defines a structure or a skeleton for an Experiment, which is referenced at the time of Experiment submission.

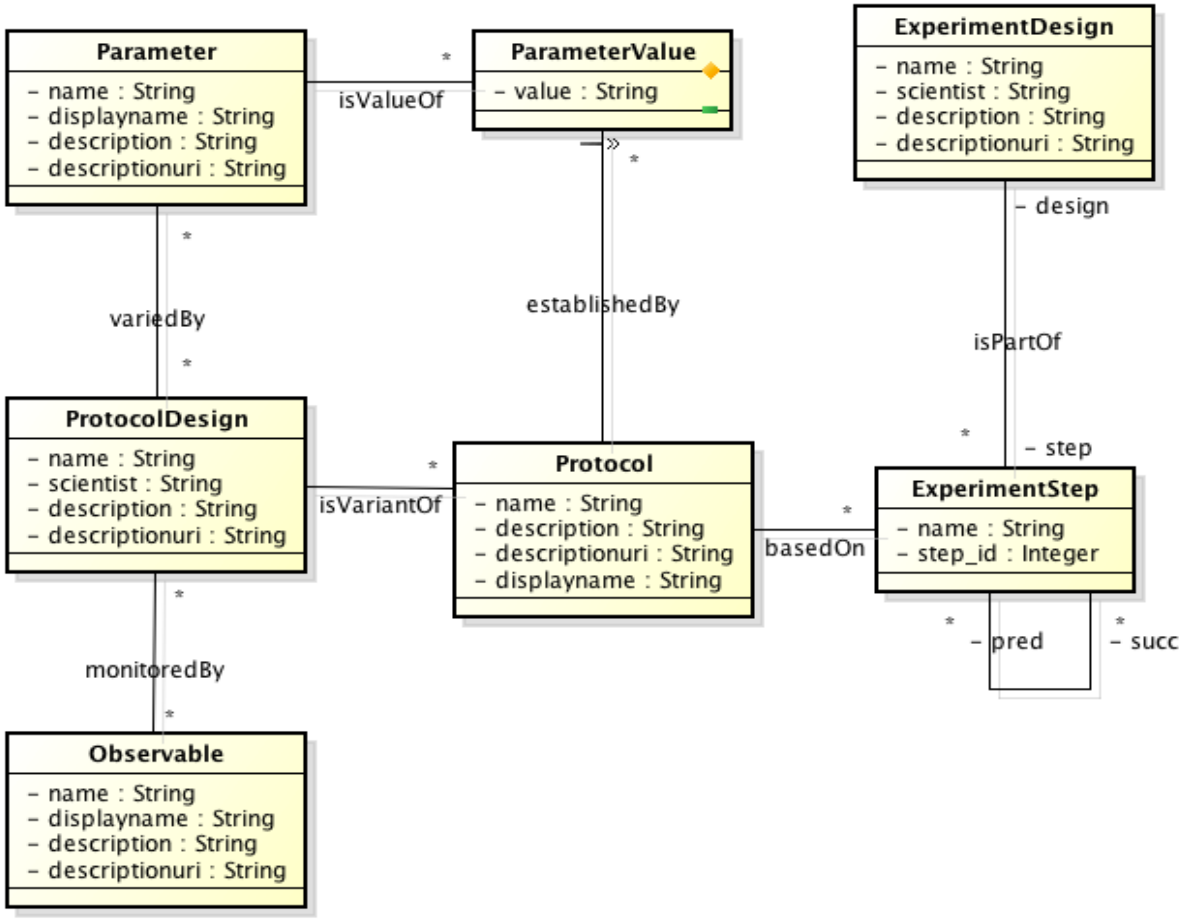

Figure 6: UML Class diagram for ExperimentDesign (Courtesy Glycomics Group)

CHAPTER 6

MAINTAINING UNIQUENESS

As stated before, we need to have a check in place, which checks for the uniqueness of data before saving it to a database. Whenever we are trying to submit an experiment design through GDaTM, there might be a possibility that the user is submitting the same experiment design with a different name. When we analyze this problem we find that this is a problem of Graph Isomorphism as the experiment design is organized as a series of directed acyclic steps, which are nothing but a Directed Acyclic Graph (DAG). For example here is a flowchart of an experiment design.

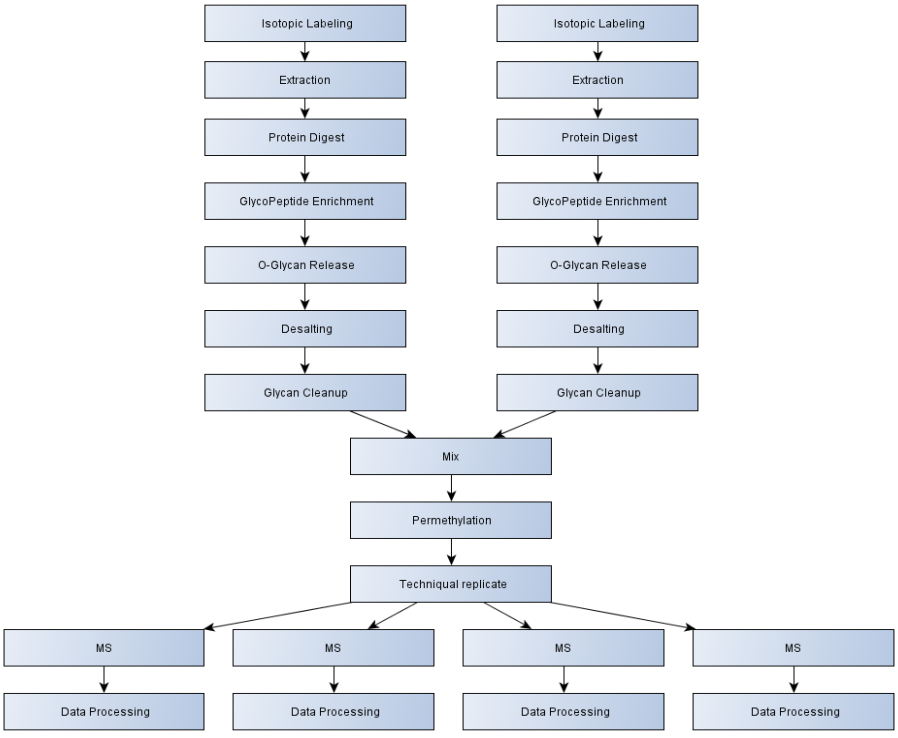

Figure 7: Static IDAWG experiment design (Courtesy Glycomics Group)

6.1 Graph Isomorphism

Given two vertex-labeled, directed graphs, G (V, E, l) and H (V′, E′, l′), they are said to be isomorphic, if there exists a bijective function f:V → V′ such that l(v) = l(f(v)) for v ∈ V and (u,v) ∈ E iff (f(u),f(v)) ∈ E′ [41]. We came up with an algorithm for Graph Isomorphism but when this algorithm was tested with the pre-existing one i.e. Nauty [30], our algorithm was outperformed by Nauty. Let us discuss Nauty and our GraphIso in a little more detail. Although this is not the focus of this thesis, we are giving general idea about both the algorithms.

6.1.1 Nauty

An automorphism of a graph is a permutation of the vertices that preservs the set of edges. In Figure 8, if we look at graph G we can interchange vertex labels 0,1 and interchange vertex labels 2,3. This rearrangement preserves the edge set, i.e., 2 is adjacent to 5 before and after, while 0 is not adjacent to 4 before or after. As shown in Figure 7, the vertex pairs (0 1)(2 3) represent an automorphism [29, 30].

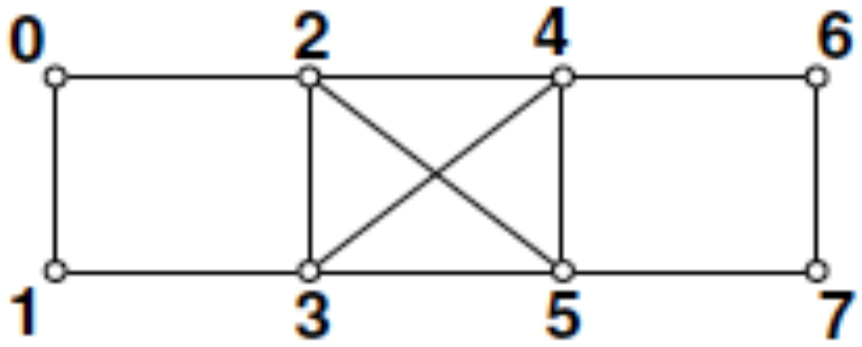

Figure 8: Graph G (Courtesy Nauty Documentation)

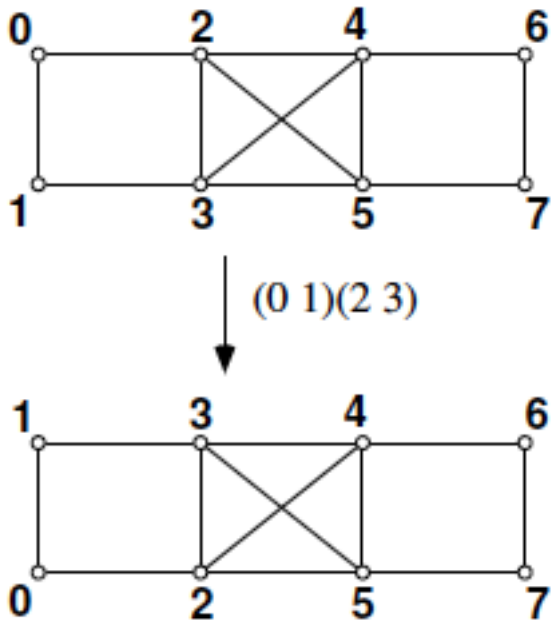

Figure 9 Graph G with automorphism (Courtesy Nauty Documentation)

Another aspect of Nauty is canonical labeling. Canonical labeling is an operation of placing the vertices in a way that does not depend on their physical arrangement. Graphs that are isomorphic (the same except for vertex labels) become exactly the same after canonical labeling (canonizing). As shown in Figure 10, two graphs, i.e., G and H are isomorphic but not identical. When we canonize the two graphs they become identical. If the two graphs (G and H) were not isomorphic then after canonizing they would not have been identical. From this inference Nauty algorithm is deduced [29, 30].

Theorem: "An isomorph of a graph G is a graph with vertex set [n] that is isomorphic to G. A canonical isomorph function assigns to every graph G an isomorph C(G) such that whenever H is isomorphic to G we have C(H) = C(G). We call C(G) the canonical isomorph of G" [30]

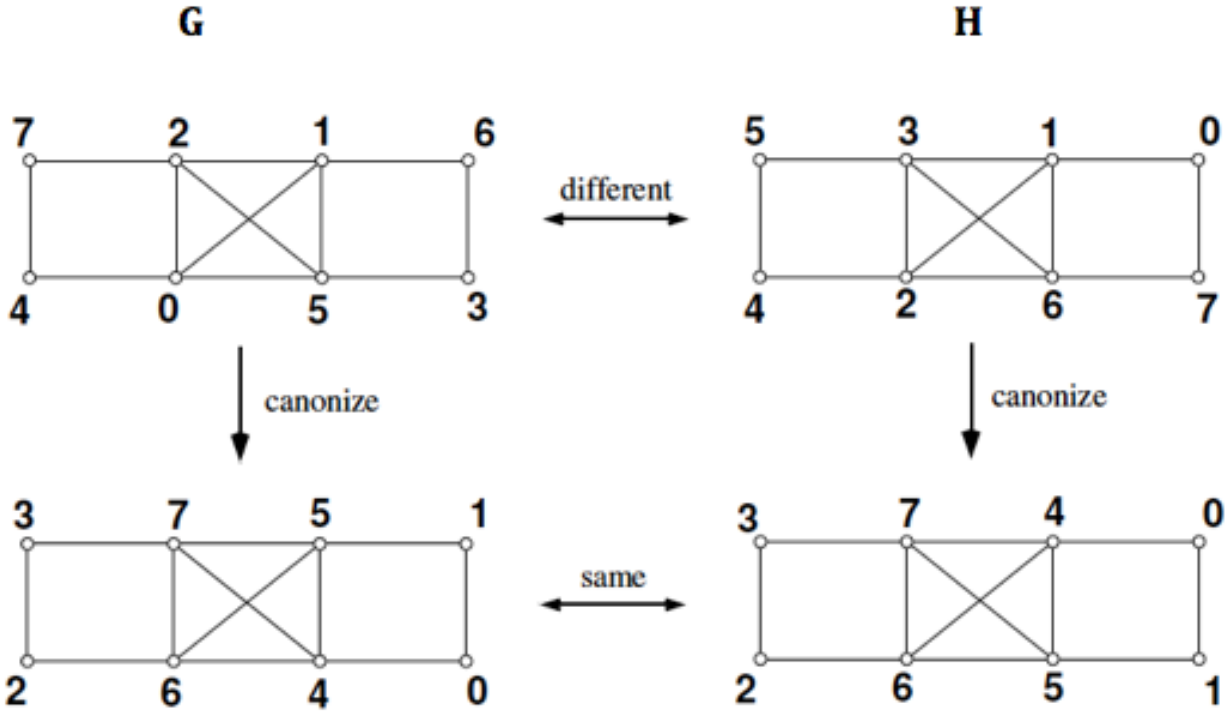

Figure 10 Canonical Labeling (Courtesy Nauty Documentation)

6.1.2 GraphIso

Before we discuss GraphIso we need to understand a concept called dual graph simulation [53], in brief.

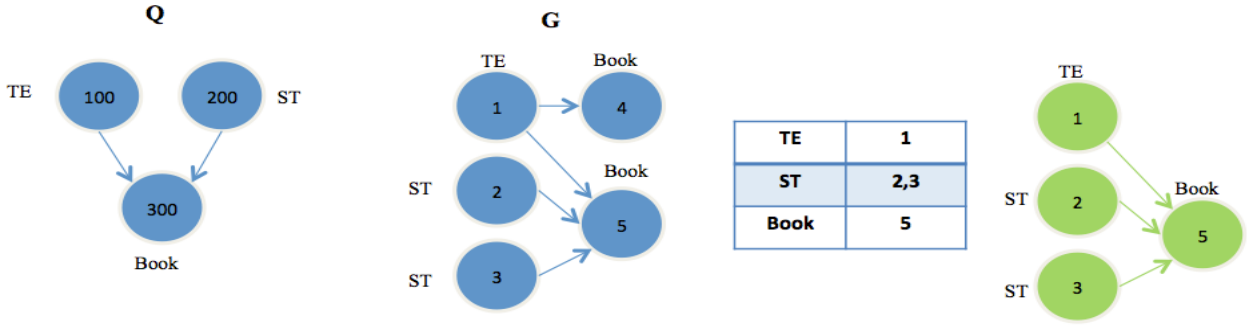

Figure 11: Dual Graph Simulation

A vertex-labeled, directed graph Q (V, E, l) is a dual simulation to graph G (V′, E′, l′), if there exists a multi-valued function φ: V→ $2^{V'}$, such that there is a label match, a child match and a parent match. As shown in Figure 11, Graph Q is a query graph and graph G is a data graph. Circles in the above Figure denote vertices. Integers inside the circle are the vertex numbers and the strings TE, ST and Book denote labels, which are represented beside or over the vertex. Dual graph simulation can be adopted as a graph pruning technique for sub-graph isomorphism. In dual simulation we look at the parents and children for a particular vertex in the query graph and try to find this pattern in the data graph. For example, if we look at query graph Q we find that the label TE has a vertex, which has a child, labeled book. Now, if we look at vertex 1 in Figure 11, we can infer that vertex one has no parent but has two children, i.e., vertex numbers 4 and 5 both bearing label book. Based on our inference for graph G for vertex 1, we conclude that vertex 1 is a probable match to vertex 100 in query graph. Similarly, we follow this approach for all the vertices in graph G and make a set of all the probable matches. Our algorithm for Graph Isomorphism is for two DAG's *G* and *H*, where a vertex corresponds to an experiment step. *#V(G)* and *#V(H)* represents numbers of vertices in *G* and *H,* respectively. DualIso calls dual graph simulation [31].

if *#V(G) == #V(H)* **then**

    return dualIso(*G*, *H*)

else

    return false

endif

CHAPTER 7

IMPLEMENTATION

GDaTM's implementation is very straightforward and follows a modular approach. As shown in Figure 12, GDaTM is implemented in three separate modules, which are integrated as a single unit.

1) *om* (Object Model), which again is subdivided into four components, i.e., ExperimentDesign, Experiment, SourceSample and ProtocolDesign.
2) *io* (Input Output), this module holds reader and writer classes for the models in *om* package
3) *util* (Utility), this module has the REST client which invokes related web services on the GlycoVault.

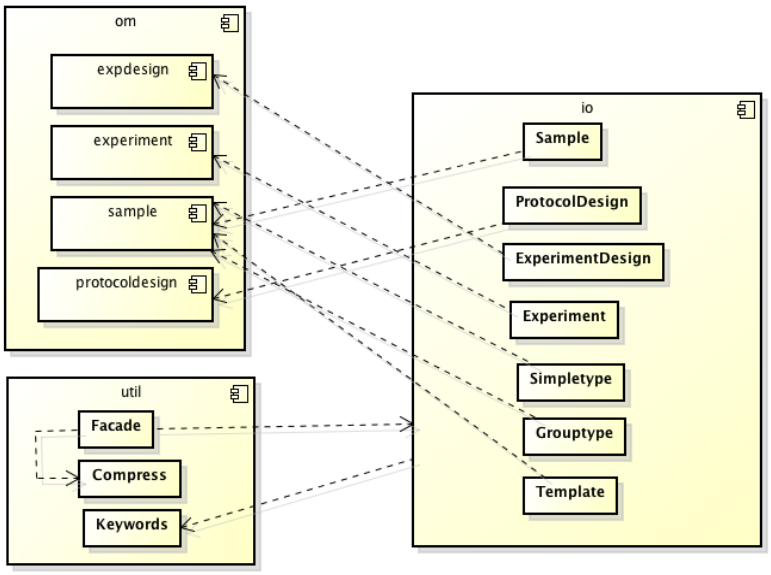

Figure 12: Component Diagram of GDaTM

Currently, GDaTM invokes the following services mentioned in Table 3. The services are invoked on the GlycoVault following the requests type mentioned in the column “Method” in the table. As shown in the above diagram, GDaTM follows a Façade pattern. The Façade class in the util package contains all the methods that are responsible for invoking the services on GlycoVault. Currently, for the serializing and deserializing we use the simplejson API.

Web Service	Url	Method	Input	Output
Enter experimental data	**/data/upload	POST	Compressed JSON	JSON confirmation
Retrieve Experimental Data	**/data/download	GET	Exp_id or Experiment Name	JSON with Experimental Data
Enter Experimental Design	**/expdesign/upload	POST	Compressed JSON	JSON confirmation
Retrieve ExperimentDesign	**/expdesign/download	GET	ExpeimentDesign id or name	JSON with a ExperimentDesign
List of Experiment Designs	**/experiment design/list	GET		JSON with list of ExperimentDesign(s)
Enter ProtocolDesign	**/pd/upload	POST	Compressed JSON	JSON confirmation
Retrieve list of protocols	**/protocoldesign/list	GET		JSON with list of Protocol(s)
Retrieve ProtocolDesign	**/pd/download	GET	ProtocolDesign id or name	JSON with a ProtocolDesign
List of Observables	**/observable/list	GET		JSON with list of Observable(s)
List of Parameters	**/parameter/list	GET		JSON with list of Parameter(s)
List of Samples	**/sourcesample/list	GET		JSON with list of SourceSample(s)
Retrieve Sample	**/source_sample/download	GET	SourceSample id or (name,biorepid,techrepid)	JSON with a SourceSample
Enter Sample	**/ source_sample /upload	POST	Compressed JSON	JSON confirmation
Enter GroupType	**/group_type/ upload	POST	Compressed JSON	JSON with a SourceSample
Retrieve GroupType	**/ group_type /download	GET	GroupType id or name	JSON with a GroupType
Retrieve SimpleType	**/simpl_type/ download	GET	SimpleType id or name	JSON with a SimpleType
List of GroupType	**/grouptype/list	GET		JSON with list of GroupType(s)
List of SimpleType	**/simpletype/list	GET		JSON with all SimpleType(s)
Enter PhysicalObjectTy	**/ physicalobject	POST	Compressed JSON	JSON confirmation

pe	_type /upload			
Retrieve PhysicalObjectType	**/physicalobject_type/download	GET	PhysicalObjectType id or name	JSON with a PhysicalObjectType
List of PhysicalObjectType (s)	**/physicalobjecttype/list	GET		JSON with list of PhysicalObjectType(s)

Table 3: List of web service (Note ** signifies the prefix of url for example "http://localhost:8080/glycovault-0.0.1/service")

Let us discuss a few of the important services briefly.

7.1 Sample Service:

Upload Sample: GDaTM invokes ***/source_sample/upload* service on GlycoVault. This service enables GDaTM to submit a SourceSample. This service is a POST request and the UML modeling for the Sample is shown in Figure 4. Figure 13 shows the JSON representation of the sample, as we can see in the Figure 13, SourceSample can contain multiple descriptors related to it where each descriptor can be of type GroupDescriptor or SimpleDescriptor. In Figure 13 if we look at the object "value" it can be of two types, i.e., a String or a JSONObject. The String representation means LiteralDescriptor and JSONObject notation means the OntoDescriptor.

Download Sample: Now that we have the capability of sending a sample, we should also have the capability of retrieving the same sample, which we have submitted. A SourceSample can be retrieved either using its id or its name with biorepid (biologicalreplicationalid) and techrepid (technicalreplicationalid).

Retrieve List of Samples: This service lets us retrieve all the samples, which are already, present in GlycoVault.

```
{
   "techrep_id": "2",
   "display_name": "sample_displayname",
   "biorep_id": "1",
   "location": "sample_location",
   "description": "sample_description",
   "name": "sample_name",
   "description_uri": "sample_description_uri",
   "po_type": "Gene",
   "active": true,
   "descriptors": [
      {
         "value": [
            {
               "value": {
                  "name": "name",
                  "identifier": "identifier",
                  "namespace": "namespace"
               },
               "type": "Tissue",
               "uri": "uri"
            },
            {
               "value": "12",
               "type": "Tissue",
               "uri": "ldesc"
            }
         ],
         "type": "Disease",
         "uri": "group_uri"
      },
      {
         "value": "13",
         "type": "Tissue",
         "uri": "ldesc"
      }
   ],
   "onto_uri": "sample_sample_onto_uri"
}
```


Figure 13: JSON representation of a SourceSample.

7.2 ExperimentDesign Service

Upload ExperimentDesign: GDaTM invokes **/expdesign/upload service on GlycoVault which enables GDaTM to transfer a experiment design in JSON. The UML for this JSON representation is as shown in Figure 6. In the JSON representation shown in Figure 14, ExperimentDesign is represented as series of ExperimentStep(s). These ExperimentStep(s) contain exactly one Protocol, which is based on a ProtocolDesign, and the Protocol can also have associated Parameter(s) to it. The Parameter(s) and the ProtocolDesign, which a Protocol refers to, should already be present in GlycoVault.

Download ExperimentDesign: GDaTM invokes **/ expdesign /download service on GlycoVault enables GDaTM to download a specific ExperimentDesign. The download service provides two ways of invoking a download service, i.e., if we know the id of the ExperimentDesign we can use the id to download the experiment design or if we know the name of the experiment design we can use that as well.

Retrieve List of ExperimentDesign: GDaTM invokes **/expdesign/download service on GlycoVault which enables GDaTM to download all the ExperimentDesign(s) which are already in GlycoVault.


```
{
   "descriptionuri": "desc_uri",
   "description": "expdescription",
   "name": "expdesignname",
   "steps": [
      {
         "protocol": {
            "descriptionuri": "protocldesriptionuri",
            "description": "protocldescription",
            "name": "name",
            "parametervalues": [
               {
                  "name": "paramval1",
                  "value": "20"
               }
            ],
            "protocoldesign": "protocoldesign",
            "displayname": "displayname"
         },
         "stepid": 1
      },
      {
         "protocol": {
            "descriptionuri": "protocldesriptionuri2",
            "description": "protocldescription2",
            "name": "name2",
            "parametervalues": [
               {
                  "name": "paramval2",
                  "value": "22"
               }
            ],
            "protocoldesign": "protocoldesign2",
            "displayname": "displayname2"
         },
         "precededby": [
            1
         ],
         "stepid": 2
      }
   ],
   "scientist": "scientist"
}
```


Figure 14: JSON representation of ExperimentDesign

7.3 PhysicalObjectType Service

Upload PhysicalObjectType*:* Enter ProtocolDesign is a POST method and is implemented as **/pt/upload in GlycoVault. Figure 15 below the JSON representation for PhysicalObject type, which can be of type GroupType or a SimpleType; the UML model for PhysicalObjectType is shown in Figure 4. As we can see in the Figure 15 the JSONObject in the JSONArray "mandatoryAttributes" represents a GroupType and the JSONObject in the JSONArray "optionalAttributes" represents a SimpleType. If we look at the "mandatoryAttributes" which is a JSONArray and hold mandatory and optional JSONArray which is nothing but a GroupType holding several SimpleType(s) in it.

Download PhysicalObejctType **pt/download/ is implemented as a GET method which can be invoked in two ways either we can provide the name of the PhysicalObjectType or its Id.

Retrieve All PhysicalObjectTypes It is a GET method which gets all the PhysicalObjectTypes present in GlycoVault.

```
{
   "optionalAttributes": [
      {
         "unitUri": "UnitUri_3",
         "unitName": "UnitName_3",
         "name": "name_3",
         "uri": "Uri_3"
      }
   ],
   "display_name":
"physicalobjecttype_displayname",
   "mandatoryAttributes": [
      {
         "optional": [
            {
               "unitUri": "UnitUri_2",
               "unitName": "UnitName_2",
               "name": "name_2",
               "uri": "Uri_2"
            }
         ],
         "mandatory": [
            {
               "unitUri": "UnitUri_1",
               "unitName": "UnitName_1",
               "name": "name",
               "uri": "Uri_1"
            }
         ],
         "name": "GTname",
         "uri": "GTUri"
      }
   ],
   "description": "physicalobjecttype_description",
   "description_uri": "physicalobjecttype_desc_uri",
   "name": "test"
}
```


Figure 15: PhysicalObjectType representations

7.4 ExperimentData Service

Upload Experiment: This service has the URI **/data/upload which is implemented as POST request. It can be considered as the most important JSON representation, i.e., the experimental data submission. The UML representation for this JSON representation is shown in the Figure 5.

To be able to successfully upload experiment data, we need to make sure we have all the supporting meta data for the experiment data already present in GlycoVault. For example, we should have the corresponding ExperimentDesign to which the experiment data refer to be present in GlycoVault. If we look at the JSON representation of experiment with its data as shown in Figure 16, we can deduce the following observations. An experiment is carried out in a series of steps called as a "Task" and these tasks are usually connected to each other in a way where the output of the one task is the input of succeeding task or an input to a task can be seen as output from several tasks. All the outputs in the form of MolecularObject or BiologicalSample are represented at the experiment level. Values produced by a task can be of types Composite, Scalar or Vector. Rules defining Values is discussed in the Appendix A of this thesis.

Download Experimental Data: This method has URI **/data/download and is implemented as a GET request. This method is responsible for retrieval of experiment data based on the id or experiment name.

```
{
   "end_date": "09/21/2014",
   "name": "name",
   "design_name": “experimentDesignname”,
   "molecules": [
      {
         "ref": 3,
         "display_name": "displayname",
         "synonym2": "synonym2",
         "synonym": "synonym",
         "description": "description",
         "description_uri": "description_uri",
         "name": "gog18",
         "onto_uri": "onto_uri"
      }
   ],
   "start_date": "09/21/2014",
   "samples": [
      {
         "ref": 1,
         "techrep_id": 1,
         "biorep_id": 1,
         "name": "sourcesample"
      },
      {
         "ref": 2,
         "based_on": 1,
         "name": "derivedsample"
      }
   ],
   "tasks": [
      {
         "input": [
            1
         ],
         "values": [
            {
               "vector": [
                  {
                     "scalar": "15",
                     "name": "retention_time",
                     "is_value_of": 81,
                     "has_type": "int"
                  }
               ],
               "name": "retention_time",
               "is_value_of": 81,
               "has_element_type": "int"
            },
            {
               "describes": 3,
               "name": "peaklist",
               "is_value_of": 1000,
               "composite": [
                  {
                     "vector": [
                        {
                           "scalar": "15",
                           "name": "retention_time",
                           "is_value_of": 81,
                           "has_type": "int"
                        }
                     ],
                     "name": "retention_time",
                     "is_value_of": 81,
                     "has_element_type": "int"
                  },
                  {
                     "scalar": "0.16",
                     "name": "delta",
                     "is_value_of": 811,
                     "has_type": "float"
                  }
               ]
            }
         ],
         "name": "task1",
         "step_id": 11,
         "scientist": "scientist",
         "output": [
            2
         ]
      }
   ]
}
```


Figure 16 JSON representation for Experiment Data

CHAPTER 8

EXPERIMENTS & META ANALYSIS

8.1 Experiments

We have conducted an experiment on a machine with the following specifications: Processor**:** 2.4 GHz Intel Core i5, **Memory:** 8 GB 1600 MHz DDR3, **Java VM**: -Xmx7068m. We have performed these experiments over the network with the router having Ethernet LAN speed 100BASE-T. We have compared the two known data transfer MIME (Multipurpose Internet Mail Extensions), is an "extension which allows people to use the protocols to exchange different types of data over the Internet", types, i.e., "application/octet-stream" and "multipart/form-data" [11, 56]. The difference between "multipart/form-data" is it is an HTTP request that HTTP clients construct to send files and data over to an HTTP Server. Browsers and HTTP clients to upload files to the server commonly use it. On the other hand "application/octet-stream" is a MIME attachment where the content type is sent over the HTTP as a binary file [51, 52]. To elaborate we have tested multipart data without compression, multipart data with compression, streaming without compression and streaming data with compression. In Figure 17, the *X*-axis represents data in MB and the *Y*-axis represents time in milliseconds (ms), the values in the *Y*-axis are averaged over three runs for each data, which is a JSON value. As we can see from the chart, streaming with compression performs better consistently and when the data starts to grow it gets even better. Multipart data transfer is slow because for a multipart transfer the output JSON has to be written into a file before it can be sent. This process of writing the file to the disk is time-consuming and as the data grows the time taken to write a file to a disk will

increase, affecting the performance of multipart data transfer. Streaming with compression not only proves to be faster but also eliminates the overhead of file management. In our experiments we found that the data transfer with streaming is significantly affected by the compression ratio for the data.

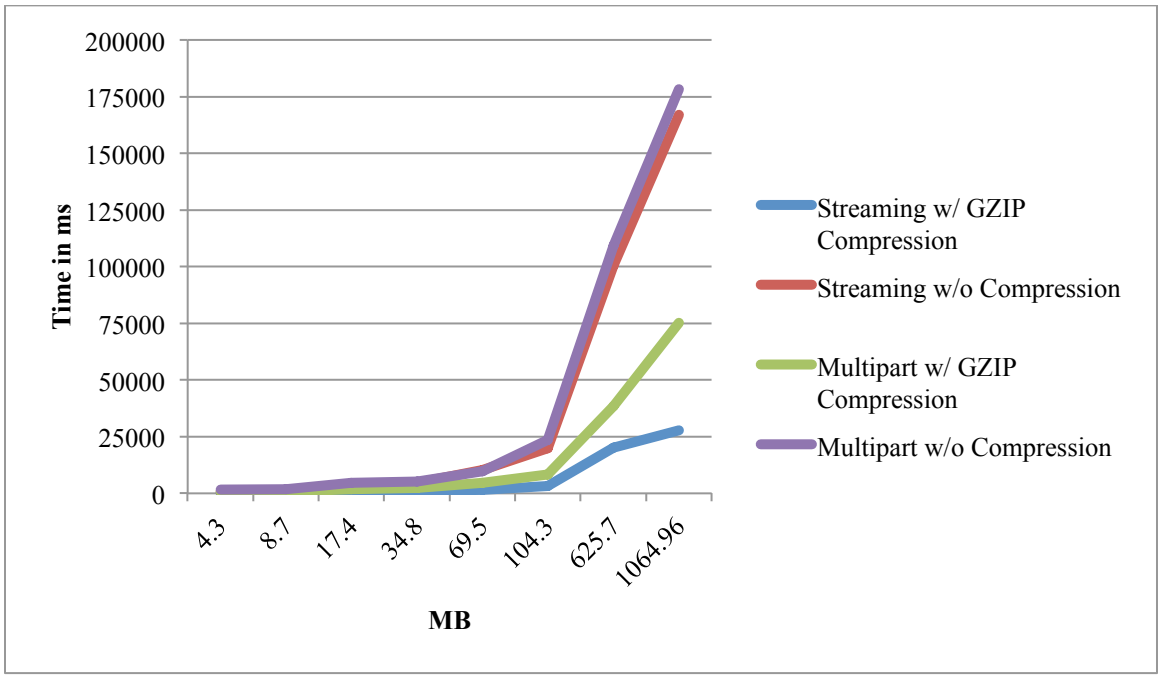

Figure 17: Comparison of various ways of data transfer

We have also compared PostgreSQL with MongoDB and the results clearly indicate PostgreSQL is outperformed in storing JSON files. We have experimented with traditional MongoDB API but it restricts its BSON internally to 16 MB. There is an API called GridFS, which is embedded in MongoDB and is used to insert BSON sized more than 16MB. GridFS divides a file into parts, or chunks, and stores each of those chunks as a separate document. By default GridFS limits chunk size to 255 KB. GridFS uses collections to store files. When we query a GridFS store for a file, the chunks are reassembled to generate a response to the query [32]. In Figure 18, we have shown the database insertion comparison with MongoDB, MongoDB/GridFS and PostgreSQL

where the input data is in the form of JSON. As shown below, clearly MongoDB's GridFS API for handling large sized JSON documents provides the fastest insertions. Note, the parsing time for JSON data being sent to PostgreSQL increases progressively. With the size of JSON for 4.3 MB it is around 6.3% and for 8.7 MB JSON it is 10.3% even after subtracting JSON parsing time from MongoDB and PostgreSQL, MongoDB is faster.

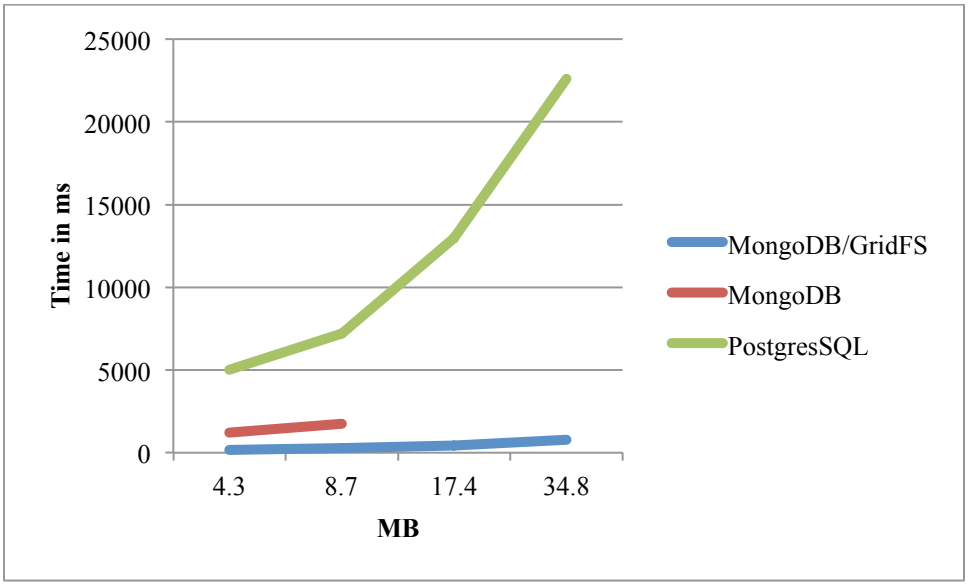

Figure 18 Performance of MongoDB against PostgreSQL in data insertion

Now, if we look at Figure 19, we have compared database retrievals for the same data i.e. JSON data, which we have inserted earlier. The experiments indicate that MongoDB's GridFS API is not only fastest in inserting data but also in retrieval of the same data. PostgreSQL is the slowest in retrieval of the data. Part of the reason for PostgreSQL slowness is because of all the relationships it has to traverse in order to build the POJO's, which are eventually written as their JSON representation and sent back in the response.

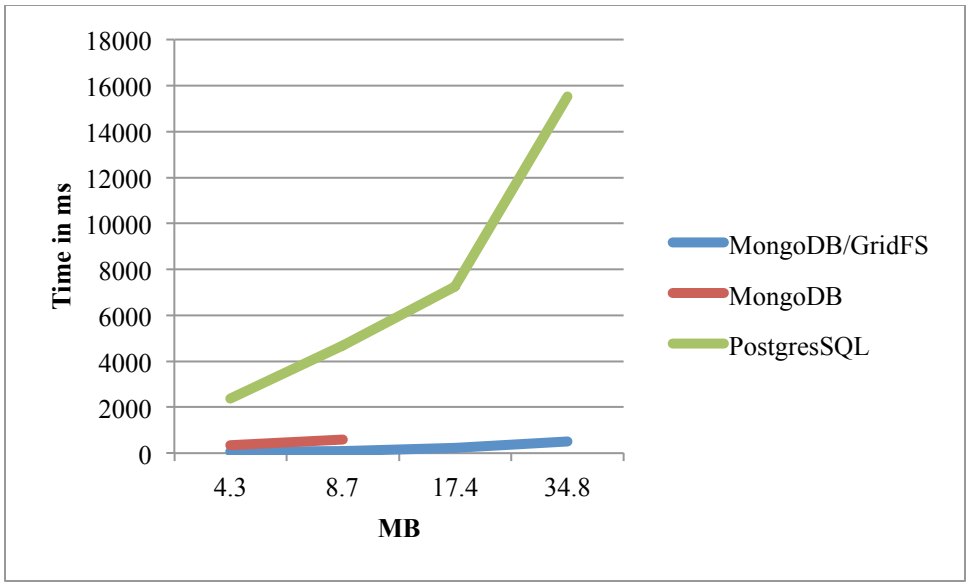

Figure 19 Performance of MongoDB against PostgreSQL for data retrieval

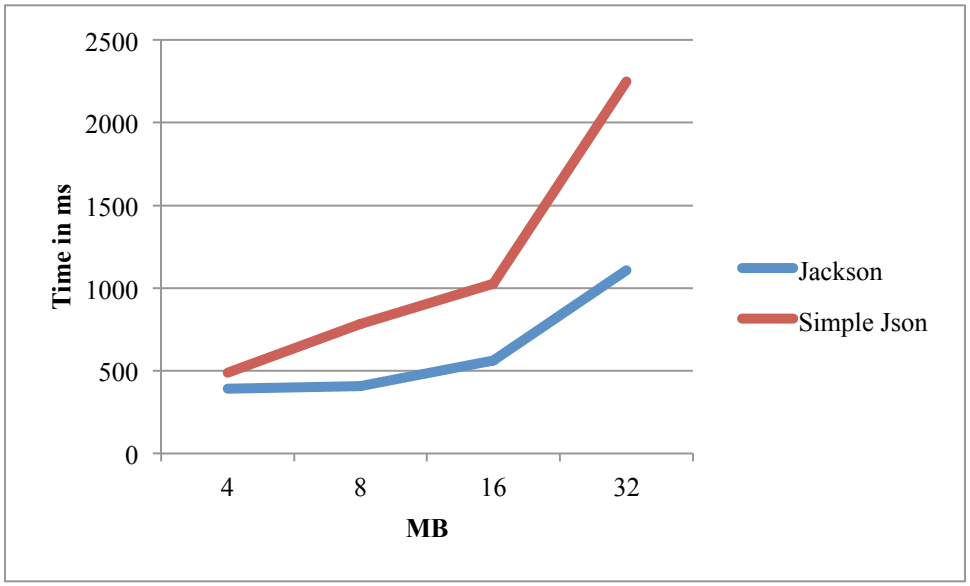

Figure 20: Comparison of JSON Parsing using Simple JSON API and Jackson

We have also compared two of the widely known JSON parsing libraries, i.e., JSON Simple and Jackson as shown in Figure 20. We can clearly see as the size of the JSON increases Jackson's performance continues to improve. There are two available implementations of Jackson, i.e., **org.codehaus.jackson** and **com.fasterxml.jackson.** We have used com.fasterxml.json for testing of our results, as this implementation is more popular than the codehaus one. Jackson is so powerful that it can replace hundreds of lines of code for serializing and deserializing the JSON by just one line of code for serializing and deserializing.

String json = mapper.writeValueAsString(object);

This has several advantages one being we do not need to worry about the object names as Jackson takes care of that. This is not the case if we use simple JSON for serializing and deserializing, as we need to write each and every object and it should have same name while deserializing it. So our code is heavily dependent on the object names. Additionally, Jackson is capable of consuming and serializing JSON of much bigger size than simple JSON can actually support.

8.2 Meta Analysis

As we have mentioned before we are proposing an appropriate choice of database for a scalable data transfer module. Before we go any further, let us discuss what a transaction is. "A transaction is a set of database reads and writes with some vital properties: reads should not be affected by writes from other transactions (Isolation), all the writes should succeed or fail together (Atomicity), and writes which are successful should be permanently stored (Durability)"[45]. Each of these is one of the ACID guarantees. Any application, which supports multiple clients concurrently accessing resources, needs transactional capabilities. Serializability is a very strong form of isolation, however its not concurrency supportive [45]. To enable more

concurrency we have to relax some of the isolation constraints. Any database transaction that happens through a service can be considered reliable only when transactions are preserved. More so there is an immense debate going on between NoSQL and scalable relational databases. Both data stores have their own advantages and disadvantages. RDBMS is completely structured way of storing data, while the NoSQL is less structured way of storing the data. Many NoSQL databases compromise ACID properties of the transactions in order to achieve certain benefits such as partition tolerance, performance, load distribution or to scale linearly which requires addition of new hardware. There are few scalable RDBMS, which are cluster based, but they have small scope operations and small-scale transactions. The notion that RDBMS prevents scalability and performance is not completely true, however, it is said that RDBMS is at least capable of performing transactions in a distributed environment. We also have to take into consideration that many NoSQL databases also avoid or make it impossible to execute a query with large scope. Many NoSQL systems simply avoid large scope transactions. One of the classic examples is MongoDB. One aspect, which we compromise in MongoDB is multi-collection (table) transactions. Modifications in MongoDB can only work against a single document. For example, if we need to debit persons A's account and credit this amount into persons B's account at the same time - we cannot do it unless these two accounts exist in the same document which is highly unlikely. As argued by many people, among the ACID properties, Isolation is one of the key factors in deciding which database to pick. Isolation is one of the properties, which is relaxed to have a scalable database, which can support large number of concurrent transactions [13, 26, 34, 35]. There are generally known 8 levels on isolation (Read Uncommitted, Read Committed, Monotonic View, Cursor Stability, Repeatable Read, Snapshot Isolation, Serializable), the most common of which are shown in Table 4 [37]. The choice of the

data store really depends on what problem we are trying to model and what isolation levels we need.

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Serializable	No	No	No

Table 4: Different levels of isolation (Courtesy PostgreSQL)

Lets us discuss a theorem, which is known as the CAP theorem. Eric Brewer concluded that a distributed system could not simultaneously provide all three of the following desirable properties:

- **"C**onsistency: A read sees all previously completed writes.
- **A**vailability: Reads and writes always succeed.
- **P**artition tolerance: Guaranteed properties are maintained even when network failures prevent some machines from communicating with others." [23]

Many researchers believe that for big data needs ACID constrains are too strict and they have shown interest towards BASE (**B**asically **A**vailable **S**oft State **E**ventual Consistency).

Basically Available: This constraint states that the system guarantees the availability of the data but, the response to a query could be 'failure' to obtain the requested data or the data fetched

may be in an inconsistent or changing state. “For example we are waiting for a check to clear in your bank account” [18, 12, 44].

Soft state: This constraint states that the system could change over time and there will be times where without any input data there might be changes going on in the system due to “eventual consistency”, thus making the state of the system always “soft” [18, 12, 44].

Eventual consistency: This constraint states that the system will become eventually consistent once it stops receiving data. The data will eventually propagate to all the places where it should exist, although the system will be continuously receiving the data. It is not necessary to check the consistency of every transaction before the system moves onto the next one [18, 12, 44].

Based on the discussion above, we can possibly think of four combinations of how we can choose the databases. First, we can think of just going with a RDBMS. In this scenario, we will achieve full ACID compliance for the transactions but we tradeoff scalability and slow database interactions as the data size increases. The second scenario can be where we decide to go with only a NoSQL data store. In this case we will achieve high scalability and faster interaction with database but may compromise some of the ACID properties of a transaction. The third scenario could be where we decide to use a scalable RDBMS we can achieve higher scalability in this and yet maintaining the ACID properties for a transaction. The fourth possibility would be if we could design a system in which we could avail benefits of both the databases, i.e., an RDBMS and a NoSQL data store. In such a system, where we deal with the exchange of large documents, we could simply put them in the NoSQL data store instead of parsing and putting them in the RDBMS, which is a very time consuming process. We could still maintain all our relationships and keys in our RDBMS for the meta data needs.

Reliability

Many researchers have argued over reliability in REST. We can achieve reliable data exchange using HTTP easily at the application level. The guarantees provided by TCP are very dependable and are highly trustworthy. Many researches have expressed concerns related to reliability of REST. One of the naïve approach could be to try to resend the data if we did not succeed in the first attempt. HTTP GET, PUT and DELETE operations have the same effect if it is called more than once with the same input parameters because they will be dealing with the same resource but the POST method creates new resources. Sending the unique ID in the response message informing the client that the resource is already there can prevent multiple POSTs of the same data. The ID returned to the client as a response can be used for further communications, hence preventing duplication of same data. Delegating responsibility of generating the message ID's to the client will not be a good solution; as suggested by Paul Prescod's: "Client should POST to a URI asking for a unique server-generated message ID to which the server returns an HTTP location header pointing to a newly created URI and where the client can POST their data" [6, 36].

8.3 Performance Tuning

In this section, we will explore some of the newer technologies, which are making their mark to solve some of the currently existing issues. These technologies not only are promising but also demonstrate how some of the problems can be solved more efficiently and effectively. The Apache Thrift software framework [46], is used for developing scalable services across variety of platforms such as C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi and other languages. Thrift supports multiple protocols other than HTTP. For faster and efficient parsing API, we have a JSON parser called as

Jackson and its performance comparison is already shown in Figure 20. It can provide in memory representation of a JSON document similar to XML DOM (Document Object Model). It also provides an alternative in which it can convert JSON into POJOs based on property accessor or annotations. It is inspired by JAXB [57]. Nowadays Postgres-XL has made its mark, which is a scalable open source PostgreSQL database cluster. It is fully capable of ACID properties and yet provides a built-in MPP (Massively Parallel Processing) capability, which allows us to run queries on large data sets [47]. In today's world, if anyone thinks about the Internet all we can think is two protocols, which dominate communication over the Internet, are TCP and HTTP. These days web pages transferred over HTTP are much different than they were two decades before. Nowadays, web pages are comparatively bigger than ever before and create web latency. SPDY is an experimental protocol designed for a faster web. It is an improvement over the HTTP which can overcome some of the bottlenecks in HTTP, i.e., HTTP is client driven, provides optional compression, can serve single request per connection. There is a new protocol "Stream Control Transmission Protocol (SCTP) -- a transport-layer protocol to replace TCP, which provides multiplexed streams and stream-aware congestion control". In this study researchers have stressed the difference between traditional HTTP where usually the underlying protocol is TCP, "which uses a single persistent pipelined TCP connection vs. HTTP over SCTP, which uses a single persistent multistreamed SCTP association" [48, 49].

CHAPTER 9

CONCLUSION

Data transfer has been a long lingering challenge for researchers generating huge data sets. Data is organized in the form of documents, which are then transferred over the network. These documents are wrapped in the message headers by the protocol, which carries the document. Multipurpose Internet Mail Extensions, or MIME, defines the format of messages, which are transferred over the Internet. There are various known MIME types [56]. In this thesis, we have compared two known data transmission MIME types, i.e., Multipart and Streaming with and without compression. Based on our experimental results, we can conclude that streaming with GZIP compression performs consistently better. Based on our experimentation for the two JSON data serialization and deserialization API's, i.e., Jackson and SimpleJson, we conclude that Jackson outperforms SimpleJson not only in speed but it is also mentioned in Jackson's documentation that it can consume and produce much larger sized JSON documents than SimpleJson. We have also proposed a meta analysis for pros and cons of Relational vs. NoSQL databases. We have also conducted experiments for evaluating the performance of PostgreSQL vs MongoDB and MongoDB's GridFS by insertion and retrieval of data in the form of JSON. Based on our experimental results, we can conclude that MongoDB's GridFS performs the best followed by MongoDB and finally PostgreSQL. Our meta analysis indicates that if we are designing a system for big data transfer we should be inclined towards BASE properties rather than ACID properties as ACID imposes much stricter constraints over the transactions which are difficult to achieve when we are dealing with huge data sets. As stated by many scientists, there

is no specific solution to the problem of efficient data transfer. The solution to this problem is often tailored according to the needs of the area or community we are dealing with. A good solution in one field of study might not be of any use in the other. As mentioned above, various combinations of data transfer with choice of a data store can be drawn from this thesis. For example, someone may prefer to go with RESTful web services with streaming with compression and a NoSQL data store, while another might have a different choice for their module depending on their data transfer and modeling needs. We can conclude that if we are dealing with an application where data transfer is in the form of JSON, then if we use a combination of Relational with a NoSQL database where the Relational data store will maintain the relationships between entities and the NoSQL store like MongoDB can be used as a document store. Hence, such a system can avail benefits of both types of databases and can be scalable and at the same time provide effective querying for large documents.

REFERENCES

[1] Palankar, M. R., Iamnitchi, A., Ripeanu, M., & Garfinkel, S. (2008, June). Amazon S3 for science grids: a viable solution?. In *Proceedings of the 2008 international workshop on Data-aware distributed computing* (pp. 55-64). ACM.

[2] Fielding, Roy. "Representational state transfer." *Architectural Styles and the Design of Netowork-based Software Architecture* (2000): 76-85.

[3] Available at gzip vs bzip 2: http://tukaani.org/lzma/benchmarks.html

[4] Xiaofeng, H. "Application analysis of JSON and XML on networked data transmission." *Comput. Programm. Skills Mainten* 10 (2010): P77-P78.

[5] Nurseitov, Nurzhan, et al. "Comparison of JSON and XML Data Interchange Formats: A Case Study." *Caine* 2009 (2009): 157-162.

[6] Availble at: http://apachecon.com/eu2007/materials/Scalable,%20Reliable,%20and%20Secure%20RESTful%20services.pdf

[7] Big Data Technologies for Ultra-High-Speed Data Transfer in Life Sciences White Paper Intel

[8] Available at: http://wiki.bigdata.com/wiki/index.php/NanoSparqlServer

[9] Moniruzzaman, A. B. M., and Syed Akhter Hossain. "Nosql database: New era of databases for big data analytics-classification, characteristics and comparison." *arXiv preprint arXiv:1307.0191* (2013).

[10] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). Hypertext transfer protocol–HTTP/1.1

[11] Available at http://www.freeformatter.com/mime-types-list.html

[12] Stonebraker, M. (2010). Errors in database systems, eventual consistency, and the cap theorem. *Communications of the ACM, BLOG@ ACM*.

[13] Tudorica, Bogdan George, and Cristian Bucur. "A comparison between several NoSQL databases with comments and notes." *Roedunet International Conference (RoEduNet), 2011 10th*. IEEE, 2011.

[14] http://www.differencebetween.net/technology/difference-between-zip-and-gzip/

[15] http://www.ddn.com/products/object-storage-web-object-scaler-wos/

[16] http://www.newscientist.com/cloudup/article/in426

[17] Pritchett, Dan. "Base: An acid alternative." *Queue* 6.3 (2008): 48-55.

[18] Brewer, Eric A. "Towards robust distributed systems." *PODC*. 2000.

[19] Available at: http://www.webopedia.com/TERM/H/HTTP.html

[20] http://rest.elkstein.org/

[21] Available at: http://javarevisited.blogspot.com/2011/11/database-transaction-tutorial-example.html

[22] Big-Data Computing: Creating revolutionary breakthroughs in commerce, science, and society

[23] https://foundationdb.com/key-value-store/white-papers/the-cap-theorem

[24] Tauro, Clarence JM, S. Aravindh, and A. B. Shreeharsha. "Comparative study of the new generation, agile, scalable, high performance NOSQL databases."*International Journal of Computer Applications* 48.20 (2012): 1-4.

[25] Available at: http://www.3pillarglobal.com/insights/exploring-the-different-types-of-nosql-databases

[26] Hadjigeorgiou, Christoforos. "RDBMS vs NoSQL: Performance and Scaling Comparison." (2013).

[27] BIG DATA: SEIZING OPPORTUNITIES, PRESERVING VALUES

Available at :

http://www.whitehouse.gov/sites/default/files/docs/big_data_privacy_report_may_1_2014.pdf

[28] Sato, K. "An Inside Look at Google BigQuery, White paper." *Google Inc* (2012).

[29] McKay, Brendan D. "nauty User's Guide (Version 2.4), 2006."

[30] Practical graph isomorphism, II Brendan D. McKay, Adolfo Piperno

[31] Saltz, M., Jain, A., Kothari, A., Fard, A., Miller, J. A., & Ramaswamy, L. (2014, June). DualIso: An Algorithm for Subgraph Pattern Matching on Very Large Labeled Graphs. In *Big Data (BigData Congress), 2014 IEEE International Congress on* (pp. 498-505). IEEE

[32] http://docs.mongodb.org/manual/core/gridfs/

[33]http://www.whitehouse.gov/sites/default/files/microsites/ostp/PCAST/pcast_big_data_and_privacy_-_may_2014.pdf

[34] http://www.thegeekstuff.com/2014/01/sql-vs-nosql-db/

[35] Nance, C., Losser, T., Iype, R., & Harmon, G. (2013). Nosql vs rdbms-why there is room for both.

[36] http://home.ccil.org/~cowan/restws.pdf

[37] Available at: http://www.infoq.com/articles/eight-isolation-levels

[38] http://developer.netflix.com/docs/REST_API_Reference

[39] Mihindukulasooriya, N., Esteban-Gutiérrez, M., & García-Castro, R. (2014, April). Seven challenges for RESTful transaction models. In *Proceedings of the companion publication of the*

23rd international conference on World wide web companion (pp. 949-952). International World Wide Web Conferences Steering Committee.

[40] Cattell, Rick. "Scalable SQL and NoSQL data stores." *ACM SIGMOD Record*39.4 (2011): 12-27.

[41] Köbler, J., Schöning, U., & Torán, J. (1994). *The graph isomorphism problem: its structural complexity*. Birkhauser Verlag.

[42] http://www.grits-toolbox.org/?page_id=52

[43] Available at: http://json.org/

[44] Pritchett, Dan. "Base: An acid alternative." *Queue* 6.3 (2008): 48-55.

[45] http://searchsqlserver.techtarget.com/definition/ACID

[46] Slee, Mark, Aditya Agarwal, and Marc Kwiatkowski. "Thrift: Scalable cross-language services implementation." *Facebook White Paper* 5 (2007).

[47] Available at: http://www.postgres-xl.org/

[48] Padhye, Jitendra, and Henrik Frystyk Nielsen. *A comparison of SPDY and HTTP performance*

[49] Belshe, M., and R. Peon. "SPDY: An experimental protocol for faster web." (2011).

[50] Nimmagadda, S., Basu, A., Eavenson, M., Han, J., Janik, M., Narra, R., ... & York, W. S. (2008, April). GlycoVault: A Bioinformatics Infrastructure for Glycan Pathway Visualization, Analysis and Modeling. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on* (pp. 692-697). IEEE.

[51] https://kb.iu.edu/d/agtj

[52] http://www.w3.org/Protocols/rfc1341/7_2_Multipart.html

[53] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, “Capturing topology in graph pattern matching,” Proc. VLDB Endow.vol. 5, no. 4, pp. 310–321, Dec. 2011

[54] DDN Whitepaper Available at: http://www.ddn.com/download/resource_library/whitepapers/ddn_whitepapers/DDN-SFX-TechnicalBrief.pdf

[55] Yuan, L. Y., Wu, L., & You, J. H. BASIC, an Alternative to BASE for Large-Scale Data Management System.

[56] Freed, N., & Borenstein, N. (1996). *Multipurpose internet mail extensions (MIME) part two: Media types*. rfc 2046, November.

[57] Available at: http://wiki.fasterxml.com/JacksonInFiveMinutes

APPENDIX A

GRAMMAR FOR VALUE

Below are the grammar rules for defining different types of “Value”(s).

If the Value is:

---

1. ScalarValue with an Observable

```
{
    "has_type": "type",
    "is_value_of": "obs_id",
    "name": "obs_name",
    "scalar": "value"
}
```

---

2. ScalarValue without an Observable

```
{
    "has_type": "type",
    "scalar": "value"
}
```

---

3. Vector with element type and with an Observable

```
{
        "has_element_type": "type" ,
        "is_value_of": "obs_id",
         "name": "obs_name",
         "vector": [ Value, ... ]
}
```

---

4. Vector with element type and without an Observable

```
{
      "has_element_type": "type",
      "vector": [ Value, ... ]
}
```

---

5. Composite with element type and an Observable

```
 {
      "is_value_of": "obs_id",
       “name": "obs_name",
       "composite":  [ "Value, ... ]
  }
```

---

6. Composite without an element type and without an Observable

{

"**has_element_type**": "type",

“composite": [Value, ...]

}

Value represents a self-identifiable value or a scalar:

One of the following names must be defined:

"scalar" -- a scalar value

"vector" -- a vector

"composite" -- a composite

or

is a float, int, string for vectors with element_type "int", "float" or "string", respectively.

APPENDIX B

JSON REPRESENTAION

Below is shown a JSON representation for a SimpleType


```
{
        "unitUri":"UnitUri",
        "unitName":"UnitName",
        "name":"name",
        "uri":"uri"
}
```


Below is a JSON representation shown for GroupType


```
{
   "optional": [
      {
         "unitUri": "UnitUri_2",
         "unitName": "UnitName_2",
         "name": "name_2",
         "uri": "Uri_2"
      }
   ],
   "mandatory": [
      {
         "unitUri": "UnitUri_1",
         "unitName": "UnitName_1",
         "name": "name",
         "uri": "Uri_1"
      }
   ],
   "name": "GTname",
   "uri": "GTUri"
}
```


Figure 21: JSON representation for GroupType

APPENDIX C

JAVADOC FOR GDaTM

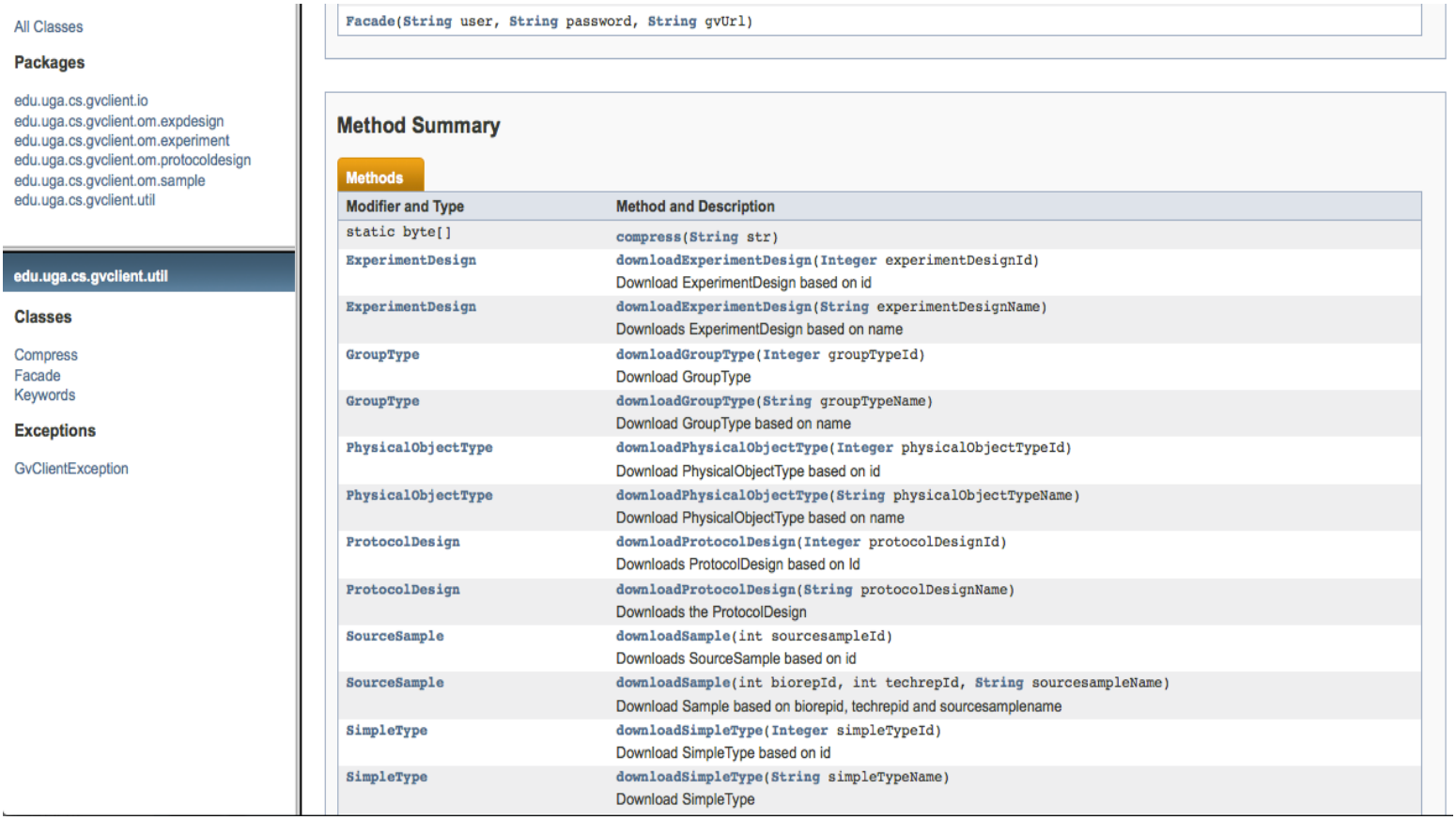

Figure 22: JavaDoc for Facade Class