Extracting Data from NoSQL Database

Ahmet Yoruk

Professor Fink

MSIN 695

December 18, 2019

Extracting Data from NoSQL Database

Introduction

NoSQL or not only SQL is the complete opposite of the traditional relational database management systems (RDBMS) that uses SQL as a query language. While SQL databases use tables or schemas to organize and retrieve data, NoSQL databases do not depend on these structures and have more flexible data models (McCreary and Kelly, 2014). Some of the downfalls of RDBMS are poor performance, inability to be scalable, and inflexibility, especially for current complex applications. Consequently, NoSQL databases have evolved into the database of choice for mainstream enterprises. NoSQL is mainly suitable for storing the unstructured data that does not fit the relational databases (Sahiet and Asanka, 2015). Examples of unstructured data include user and session data; log data, time-series data, and large objects like videos and pictures. Unstructured data is high in volume, and businesses have to identify effective ways to analyze and derive useful information that can offer them a competitive edge (McCreary and Kelly, 2014). NoSQL databases are not relational, have no explicit schema, and scale horizontally, which means the traditional tools used to extract data are rendered unusable. This thesis will analyze the challenges involved in the extraction of data from NoSQL databases and suggest potential solutions.

Relational Databases

DBMS Relational databases represent data in rectangular tables that are called relations. A single relation has a collection of titled attributes or in other words, named columns in the table. An attribute is affiliated to a domain, which is the atomic type like an integer or a string. Rows in a table are referred to as tuples (Jurie, 1992). Therefore, a relation with, say ten attributes, means each tuple or row has ten components for each of the attributes. Also, all the attributes have to be within the domain of their designated attribute. The schema is the title of the relation and its collection of attributes (Jurie, 1992). A majority of relational database management systems assure ACID transactions, which is an acronym for atomicity, consistency, isolation, and durability (Blasgen, 1982). Atomicity of transactions mean very operation is carried out entirely or not at all. Consistency involves ensuring all transactions transition the database from a single valid state to another. Isolation is ensuring the transactions do not interfere with one another. Durability means the effects of a transaction have to persevere and, therefore, cannot be lost.

NoSQL Databases

The word NoSQL had its origin in 1998 when researches used it for a particular RDBMS that did not support SQL2. The name came up again in 2009 to indicate the type of database it currently refers to (Vatika and Meenu, 2012). Today, NoSQL is used to refer to the alternative to relational databases or SQL in situations where SQ is not suitable or sufficient. Therefore, NoSQL was not coined to replace SQL but rather to complement it. Bigtable and Dynamo are two of the most groundbreaking papers concerning the NoSQL campaign (McCreary and Kelly, 2014). A lot of design choices and ideologies presented in the documents can be traced to the succeeding works.

Interestingly, groundbreakers in the NoSQL agenda have been large web corporations like Google and Amazon. NoSQL is an amalgamation of an extensive list of diverse systems which in most cases, were created to address an issue with the relational database system but did not make the cut. Typically, the NoSQL database has a non-relational data model, does not have schema definitions and scales horizontally (Van de Camp et al., 234). Also, whereas SQL maintains integrity through ACID properties, NoSQL databases are founded on CAP priorities, which stand for consistency, availability, and partition tolerance (Vatika and Meenu, 2017). Unfortunately, it is difficult to achieve all three priorities, thus users have to select two out of the three, given the Brewer's Theorem. The theorem suggests that it is not possible to attain all three of the priorities at the same time within a distributed environment. There are NoSQL databases that prioritize consistency and availability. Consistency is achieved through eventual consistency, which ensures each modification is propagated throughout the database, although it is possible to have nodes that have not been updated at a specific time (Sahiet and Asanka, 2015).

Big Data

NoSQL is the database of choice in Big Data. Supporters of NoSQL systems believe the high scalability and excellent performance make NoSQL suitable for Big Data compared to the traditional relational databases (Leclercq, 2016). Furthermore, NoSQL, such as Cassandra, MongoDB, and Redis, can store unstructured data. Several large corporations like Amazon, Google, LinkedIn, and Facebook have already incorporated Big Data NoSQL databases into their operations to address the issues associated with relational databases. An excellent example of a challenge encountered when utilizing relational databases is that it does not support the increased use of unstructured data (McCreary and Kelly, 2014).

The data processing requirements for Big Data are enormous, making the dynamic nature and cloud support of NoSQL the best choice since it can quickly and dynamically process unstructured data. Therefore, regardless of the conclusions drawn from arguments about whether to use SQL or NoSQL, the evolving and increasing business data management requirements make NoSQL a massive player in the big data movement (Scholz, 2017). Some of the examples of NoSQL databases in use include HBase for Hadoop which is used by Facebook in their messaging platform, HBase used by Twitter to generate data, storage, logs, and monitoring data in the search tool, and MongoDB is used by CERN to gather data from the particle collider. Also, both LinkedIn and Concur utilize the Couchbase. NoSQL Database for their data processing and monitoring jobs (Fowler, et. al., 2016).

Cloud Computing

Cloud computing refers to a computing infrastructure that allows access to applications from any location globally when demand arises. Some significant features of cloud computing include on-demand service, broad network access, and high scalability (Mohammed and Osman, 2012). On-demand service means it is possible to access the service without the need for human intervention. Broad network access allows access to cloud services by diverse thick, and thin client applications. High scalability refers to real-time, atomic, and rapid resource allocation. Examples of NoSQL database that can run on the cloud include Apache Cassandra, CouchDB, and MongoDB. NoSQL databases are suitable in the cloud environment because they support massive read/write loads and are highly scalable (Mohammed and Osman, 2012).

Benefits of NoSQL Databases over Relational Databases

According to Fowler et. al. (2016), there are multiple reasons to implement NoSQL rather than relational database. For instance, volume of data handled in a specific period or the distinctive quality of the data in question (Fowler et al., 2016). NoSQL databases have clear advantages over relational databases, particularly at an enterprise level. For instance, the database has high scalability since they have a horizontal scale-out structure, which makes the addition and reduction of capacity much more straightforward, faster, and in a non-disruptive way to the hardware. High scalability eradicates the high cost and complication of database shading that is essential when attempting to scale relational databases.

Furthermore, NoSQL databases perform exceedingly well, which enables the continuous reliable, fast user experience. The database is highly available and straightforward compared to relational databases that depend on primary and secondary nodes. There are networked NoSQL databases that have a master less structure that automatically allocates data equally among several resources, which ensures an app is still available for operations in case a node fails (Nasholm, 2012).

Additionally, NoSQL databases offer global availability by replicating data on several servers, data centers, or on the cloud. Distributed NoSQL databases can reduce response time and guarantee a reliable app experience despite the location of its users. Also, the distributed version substantially eases management burden synonymous with manual relational database configuration, allowing business staff to focus on other tasks. Finally, the NoSQL database can apply flexible data models. Therefore, app developers are enabled to utilize data types and queries that conform to a particular app rather than altering the app to fit the database (Nasholm, 2012). NoSQL databases allow agile app creation and non-complex interaction between the app and the database.

Types of NoSQL Data Store

There exist four significant kinds of NoSQL data stores that are classified based on their application. They include; key-value data stores, document stores, wide-column stores, and graph stores. There are multi-modal databases that utilize some amalgamation of the four types of NoSQL databases and can be used by broader assortment of applications.

Key-Value Data Stores

The key-value database, such as Redis, focuses heavily on simplification and usefulness, which improves an app’s performance by allowing ultra fast read and write processing of non transactional data (Vatika and Meenu, 2012). Stored values that are read using a key can be any form of a binary object, including message logs, video, and JSON document. This type of NoSQL model is the most flexible since the app has full control over what is stored in the value. Data is partitioned and duplicated across a cluster to ensure it remains scalable and available. Therefore, key-value stores usually do not support transactions. Nonetheless, they are very effective at scaling apps that handle fast-moving, non-transactional data.

Document Stores

Document databases like MongoDB usually store metadata such as JSON, XML, and BSON documents (Van de Camp et al. 2017). They are almost the same as key-value stores, although, for document stores, a value represents one document that stores all data associated with a particular key (Vatika and Meenu, 2012). It is possible to index popular fields in the record, which allows quick retrieval in the absence of the key. The document does not have to have a similar structure.

Wide-Column Stores

Wide-column databases such as Cassandra store data in tables with rows and columns like relational databases, though it allows a variation of the names and formats of columns between rows across the table (Vatika and Meenu, 2012). Wide-column stores cluster columns that have correlated data. Therefore, a query retrieves related data in a single operation since the columns connected to the query are retrieved. On the other hand, for relational databases, the data is likely to be in distinct rows that are stored in different places on the disk, which necessitates several disk operations to retrieve the data.

Graph Stores

A graph database like the Neo4j makes use of graph structures to store, map, and query associations (Vatika and Meenu, 2012). It offers index-free contiguity so that contiguous elements are connected without the need for an index.

Possible Solutions

Over the years, researchers have published solutions that will allow the extraction of data from NoSQL databases. These solutions can be grouped into three categories. First, the translation of SQL queries to NoSQL applications. Another proposed solution is the conversion of data from NoSQL databases into relational databases or the other way around. These two solutions highlight the importance of relational databases in the extraction of data from NoSQL databases. The final solution is the creation of a unified data access platform that can query both relational and non-relational databases at the same time. A unified solution would eliminate the need for programming and the need to convert data between the SQL and NoSQL systems. The possible solutions include SQL++, unified data access platform, and Spotfire.

Unified Data Access Platform for SQL and NoSQL Databases

Vathy-Fogarassy and Hugyak proposed developing a method that integrates the data using JSON objects, which solves the semantic and syntax issues associated with the SQL and NoSQL databases. Their research paper proposes the use of a united data access platform that is web-based and allows users to perform querying and merging from diverse RDBMS and MongoDB. The platform will give its users the ability to delineate the semantic of the source database in a user-friendly form, which would allow users to request data using attractive graphical user interfaces. Even though the present solution by the researchers required a manual resolution to semantic heterogeneity is resolved manually, the proposed platform is still a suitable reason to support partially automatic system-based mapping.

The paper is a theoretic proposal for platform that solves the heterogeneity issues of current databases systems (Vathy-Fogarassy and Hugyak, 2017). First, general data models have to be formulated in JSOON to define the significant features and relationships of the source data this addressing the structural and semantic heterogeneities of source data. The diverseness of dissimilar access approaches is handled using database adapters that utilize data manipulation functions depending on the language and configuration of source database. Their idea of creating a general data model is to address the challenge semantic and syntactical differences of both SQL and NoSQL. The reports for the general data models are called general schema (GS), which comprise of both the framework and semantic details of the source database. The GS is usually JSON objects. Vathy-Fogarassy and Hugyak opted to use JSON since it is a language-independent and open-standard format. Also, the language's metadata nature enables the easy the easy description and comprehension of the primary features of a database.

The general schemas are produced automatically from the database objects of the source database. Therefore, relational databases generate their GSs from tables and views. On the other hand, NoSQL databases are founded on multiple data models, which means they have various database objects. For instance, the collection is the database object for document stores; thus the collection helps produce the GSs (Vathy-Fogarassy and Hugyak, 2017). All GSs have a unique name that is derived from the name of the source object, whether a table, virtual tables, or collection.

Furthermore, the GS comprises many details about the source database object, such as details about the source database, the framework description of the database object, semantic mapping of database object elements, and the relationship of the object to other objects. Source database description includes details like the name and type of the data, which is required to form the connection to the server and to produce queries in the correct query language. The framework description of the database object usually includes the names and attributes of the database. The GS also records the key attributes to allow the joining of associated data (Vathy-Fogarassy and Hugyak, 2017). Semantic mapping information refers to the alias assigned to each attribute or element in the object. The pseudonyms ensure the logical name adaptation for the elements in GSs. The names can be created manually with the use of ontology-based techniques or produced semi-automatically. The aliases allow users to reference attribute minus the content of the object and database. Given the considerations when defining the aliases, a similar pseudonym means the objects are semantically identical. The description of the relationships between database objects is recorded in the GS using the referencing object that stores the details of the association.

Data Integration

According to Vathy-Fogarassy and Hugyak, the integration of data originating from diverse databases is quite tricky and involves several challenges. Nonetheless, the increased popularity of NoSQL databases and improved synthesis of different NoSQL systems have made it essential for interoperability between NoSQL and SQL databases (Vathy-Fogarassy and Hugyak, 2017). Data integration is possible on multiple abstraction levels, which also determine the appropriate technique. There are six applicable techniques, including manual integration, providing a standard user interface, creating an integration app, implementing a middleware interface, coming up with uniform data access, and making a shared data storage. Even though manual integration, standard user interface, integration app, and middleware interface have unique substantial disadvantages, they are straightforward solutions. For instance, manual data integration tales a lot of time and it is costly technique. An integration application enables access to different types of databases and returns the combined output to users, but the app can only handle the databases configured into them, which means an increase in number and type of databases makes the app quite complicated.

On the other hand, a shared user interface does not combine the data leaving the homogenization and data integration up to the users. Examples of middleware include Object Linking and Embedding Database (OLE DB), Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC). The database access drivers offer general functionalities that assist with querying and manipulating data in various DBMSs, but the drivers cannot puzzle out the structural and semantic variances of source databases (Vathy-Fogarassy and Hugyak, 2017).

Creating shared data storage and migrating all the necessary information into a new database is a complicated approach. The procedure itself has to be programmatic and cover all the ETL processes; the extraction, transformation, and loading of data (Vathy-Fogarassy and Hugyak, 2017). Data transformation is a critical step in the ETL process and involves the semantic mapping of both data types and structures from the source system into those of the target system. Some of the significant challenges of data migration include heterogeneity of syntax and semantics, data inconsistency, and the absence of management policies. These issues have resulted in manual or semi-automated data migration. The most significant shortcoming of the data migration process is that it needs storage space, and the migrated data does not automatically adopt the alteration of the source data from the source systems. Data migration has to be performed again when a user wants to make changes in the source for the stored data, which results in an expensive and time consuming undertaking (Vathy-Fogarassy and Hugyak, 2017).

The uniform data access provides consistent data integration and solves many of the drawbacks of the other techniques. The technique does not require the storage of data on the target database which eliminates the need for repeating data migration when the user needs to update the data (Vathy-Fogarassy and Hugyak, 2017). Although, there is a need for developers to handle the extraction and transformation of data from the ETL process since there is no loading phase. In contrast with data migration, just the necessary information is queried from the source database, which means it is possible to perform schema matching databases dynamically. The biggest challenge for uniform data access is handling the semantic heterogeneity of source databases. Semantic heterogeneity is a term that refers to the dissimilarities in implication and understanding of data from a similar domain. The difference in the thinking process between developers results in heterogeneity, which means they will output different models even when the end goal is the same. Logically, it is not possible to fully delineate the meaning of data in a database, and there is no categorical semantics that applies to every database (Vathy-Fogarassy and Hugyak, 2017). The challenge of conciliating schema diversity has been an important research topic in many papers, and the unified data access platform is founded on the knowledge gathered and can be executed either manually or semi-automatically.

Vathy-Fogarassy and Hugyak developed a web application that supported MySQL and MongoDB databases to show the usefulness of a unified data access platform. The web app handled the architectural differences of the source databases within a database layer. The layer dealt with the generation of database operations, depending on user requests. The benefits of using unified data access platform are highlighted when a new database is added to the platform. Users can quickly delineate the semantically sharing between the source DBMSs without having to know the internal structure of the source databases to retrieve data from them. The unified data access platform does not support joins and aggregates across data sources but gathers data from various disjointed database systems depending on the filtering choice and moves the data. Also, it is founded on a meta-model methodology, which means it addresses all the differences of the source system (Vathy-Fogarassy and Hugyak, 2017).

SQL++

SQL++ is a potential solution that has recently been published. It utilizes a new semi structured data model referred to as the SQL++ data model which is a superset of the relational data model and JavaScript Object Notation (JSON). SQL++ is founded on SQL which means its semantics and syntax closely resembles the popular SQL (Ong et al. 2014). Consequentially, the SQL++ data model is an extension of both the relation model and JSON. Thus its query language is an extension of SQL with semi-structured capabilities similar to non-relational languages. The SQL++ data model expresses relational tables as JSON formulas in such a manner that a single row is mapped into an object literal. It is possible to query and manipulate the data stored in the SQL++ data model using the SQL++ query language (Ong et al. 2014). The SQL++ query language inputs and outputs SQL++ data and can take on nested and heterogeneous forms. SQL++ is capable of handling both SQL and NoSQL databases due to its configuration and the SQL++ data model. Furthermore, the SQL++ query language can be transformed into other SQL++JSON database languages like Cassandra and MongoDB. Even though SQL++ is backward compatible with SQL, the relational trait has no dominion over the entire system (Ong et al. 2).

The SQL++ solution is quite similar to the unified data access platform since both solutions are founded on the principle that JSON is appropriate for mapping diverse data models into a single model. Nevertheless, the two solutions have significant differences. For instance, in SQL++, the source data is mapped into SQL++ data, which is then subject to all the data manipulation operations, and the result of the semantic heterogeneities is performed on the output data using the SQL++ query language (Vathy-Fogarassy and Hugyak, 2017). On the other hand, the unified data access solution performs data selection on the source systems using built-in query languages, and then output is mapped into JSON. Consequently, the data manipulation functions of detached source systems are run on JSON literals (Ong et al. 2014). Also, user requests in SQL++ are formed using SQL++ queries, while user requests in the unified solution are made on the graphical user interface.

Moreover, SQL++ includes the full functionality of SQL, while the unified solution features are not entirely all-inclusive. The authors of the unified data access platform were to extract data from diverse systems but do not support data aggregation and grouping. In other words, the purpose of SQL++ and the unified data access platform are different. SQL++

delineates a comprehensive and very expressive data querying language that can query data from heterogeneous databases, but to create the queries, users require programming knowledge. Whereas, the unified data access platform does not require any programming knowledge since the data queries and mapping of data are performed on a graphical user interface (Vathy-Fogarassy and Hugyak, 2017). Spotfire, a product of TIBCO Software, is a software platform that is used for interactive data analysis and visualization (Nasholm, 2012). Spotfire gives businesses and companies the capability to derive useful information from their data like sales data, which enables informed decision making that, in turn, generates economic value for the business. For instance, a company with multiple store locations that sell various products could adopt Spotfire to analyze their sales for each site and product, this identifying trends patterns among the customers (Ahlberg, 1996). Such information is useful, particularly when the managers have to make informed decisions such as the budget distribution for each location behaviors, determining the appropriate products that should be stocked, and other choices that improve the overall profitability of the business.

Spotfire offers a public SDK that gives app developers the necessary resources required to extend the platform. The platform supports numerous extension points such as, tools that handle the analytic tasks, data transformation that process data before it is analyzed, and data visualization, which presents the data in a user-friendly manner. Custom extensions can be created using the public Spotfire API. There are tutorials, examples, and API documentation that offers reference material on the use of the Spotfire API and public SDK (Nasholm, 2012). The extensions can be written using the .NET languages. Also, the basic unit used by Spotfire is the analysis file, which further divided into a collection of pages that comprises several visualizations. Some data visualization forms offered by Spotfire include the Charts, Scatter

Plots, Heat Map, and Box Plot. Before data visualization, data importation from the source database and data transformation of the data into tables has to be complete. The data tables are similar to relational database tablets, which have rows and columns.

Cassandra Tool

Cassandra is an accessible NoSQL database with numerous prominent users and receives professional support from third parties. The database is considered one of the better systems within the NoSQL family. Given the amount of support Cassandra gets, there is a constant release of new versions, and the variations between each version can be reasonably significant. Similar to Spotfire, developers can link with Cassandra using an API, in this case Thrift API, or a high-level client that offers an additional abstraction layer (Nasholm, 2012). In the implementation described here, all communication is done directly via Thrift. Unfortunately, high-level clients are not as mature as Cassandra which means they increase the risk of encountering blocking bugs. The process of identifying an error and fixing the issue or finding a solution takes up a lot of time. Also, each time there is a new feature on Cassandra, high-level client developers have to develop support for it before the feature can be useful to an app. The extraction of data from Cassandra database, its data models, has to be mapped onto tables using the same procedure as a unified data access platform (Nasholm, 2012).

Neo4j Tool

The Neo4j database also has an active community that provides a lot of support, making it a widely used database that translates to interest in assisting in implementation. The primary method for connecting to a remote Neo4j instance is through a REST API over HTTP (Nasholm, 2012). The REST API provides several relevant functionalities that can be used to access data in the database. Nonetheless, the REST API runs over HTTP which means there some scenarios where a query returns more data than is necessary; making communication using the standard REST API from Spotfire can be inept. Hence, the better option is to develop a server plugin that implements Spotfire explicit functionalities on the server-side. Connecting to the server plugin is also accomplished through a REST API, which is an addition of the plugin between the API and the database (Nasholm, 2012). The plugin ensures that the output contains the relevant data, and the functionalities match the Spotfire requirements. This approach optimizes efficiency even though the developer has to set up the server plugin no the server before any other steps. Also, mapping the Neo4j data model to tables by converting the attribute values into a form supported by Spotfire but in this case, only importing nodes is supported (Nasholm, 2012). To be precise, nodes are presumed to be equivalent to the rows in a table, and the node properties can be equated to the columns.

Challenges Involved in the Extraction of Data using Spotfire

Given the potential NoSQL has, especially in Big Data, data extraction challenges have to be studied and solutions identified. One of the significant challenges of NoSQL is the importation of data from NoSQL application into an analytical platform such as Microsoft BI and Spotfire (Nasholm, 2012). The issue therein lies in the conversion of the NoSQL data models into a suitable data model. The tables generated are a representation of an assortment of analogous entity instances. Hence, facilitating support for NoSQL data stores in Spotfire involved the facilitation of both extraction and important of comparable entity instances into the Spotfire tables. Three issues that have to be addressed to enable data transformation into Spotfire tables are identifying ways to allow users to specify the analogous entities and their attributes, where to extract the information, and the means to transform attributes values into supported values by Spotfire. Once the user determines that data shall be derived from SQL database, then the description of the database and its attributes is known given that the database has a single entity, and every instance is structured similarly (Nasholm, 2012). These assumptions are correct in relational databases, which makes it easy for the user to both specify the analogous entities and their attributes and extract the metadata information since the user can retrieve this information directly from the database.

On the other hand, NoSQL databases do not have explicit schemas; therefore, extracting the same information is a more significant challenge. Nevertheless, since there is a structure storing one entity, a solution for the issue is to let the user provide a full path to the structure and offer the general schema textually. For instance, making the assumption that a Cassandra CF stores a single entity and that CF columns can be mapped to table columns, it would be workable to let the user provide the CF path and columns. Though this approach is mostly not favored since the user may not have the precise structure of the database, and also there is the possibility of typing errors that would result in unwanted output. Instead, a more directed method is preferred where the schema can be deducted by sampling the database. The deduced schema can then be shown to the user, who then selects the attributes they wish to extract from the entire selection, similar to the relational database (Nasholm, 2012). Naturally, the user can modify the final schemas since the output may not be flawless due to the sampling of the database. The general method is commonly referred to as schema inference. Since sampling the whole database would be an expensive undertaking, it is suggested that schemas are deduced over smaller structures.

Identifying what should be extracted, queried, and retrieved is simple for relational databases due to SQL. But NoSQL systems usually have diverse querying interfaces that support various types of queries. Consequently, there is no one single solution to the challenge. A possibly approach is to analyze each system's interface, and inference is made from the analysis result. Also, the data should be extracted from the sampled database since the inferred schema is valid for that particular structure. Finally, the issue of data transformation for NoSQL databases is simple since it involves just the type conversion or the creation of a conversion procedure (Nasholm, 2012).

Review of Solutions

Cassandra Tool

The use of the Cassandra tool introduces the issue of sort order for the returned rows during a query. The sort order is consecutive contingent upon the data model which might have a definite meaning, and could, as a result, restrict the validity of the sample. Unfortunately, the issue is intrinsic to the query model of the database which makes it unavoidable. Nasholm suggests identifying another sampling technique, which has yet to be discovered, that avoids the sort order issue.

Nonetheless, the use of several sampling techniques is not as essential for the Cassandra tool as it is for the Neo4j tool since it is presumed that the row design in the sample are similar while the nodes in the Neo4j could model different entities (Leclercq, 2016). The use of the random partitioner for the cluster returns rows that are semantically randomly ordered as much as the MD5 algorithm allows. Also, users can write custom type converters between Cassandra and Spotfire types that can be used during the importation of data from Cassandra database to the Spotfire platform. Custom type converters have to be enabled by allowing the input DLLs. Consequentially, the research gap for the Cassandra tool is in areas related to the support of a wider variety of type and more refined type inference algorithm than the present algorithm.

Neo4j

The Cassandra and Neo4j tools are abstractly different whereby the Neo4j tool assumes that there is only one entity for each sample, while for the Cassandra tool there is more than one entity for each sampled structure. Thus, additional clustering support is crucial for the Neo4j tool. The proves of sampling and clustering for the Neo4j tool experiences two challenges, the possibility of deriving a non-representative sample and invalid clustering. Sampling errors are as a result of having a small sample size or applying the wrong sampling technique. Both clustering and sampling have to be improved to improve the overall schema inference. Unfortunately, clustering is challenging to perfect which makes it the aspect that requires most focus. Also, the sampling techniques have to be improved by incorporating multiple methods into a robust hybrid that selects accurate samples. Researchers have to develop a precise distance measure that considers more parameters and alternative seeding techniques that can be implemented. For instance, Zou et al. found a way for the WekaDB to use the Weka data mining library, which enables the storage of data used by learning algorithms similar to the MIND algorithm thus allowing scalability for massive data sets (Wang et al. 2004). The WekaDB offers a fascinating benchmark since it delivers valid employment and implementation of learning algorithms (Zou et al, 2006). Finally, there needs to be more research in techniques used in determining the optimum clusters which will highlight the real relevance of the cluster count.

Unified Data Access Platform

The unified data access platform suggests the design of a hybrid database that can store both relational and non-relational data. The hybrid database has three sections, the database layer, server-side, and the user interface.

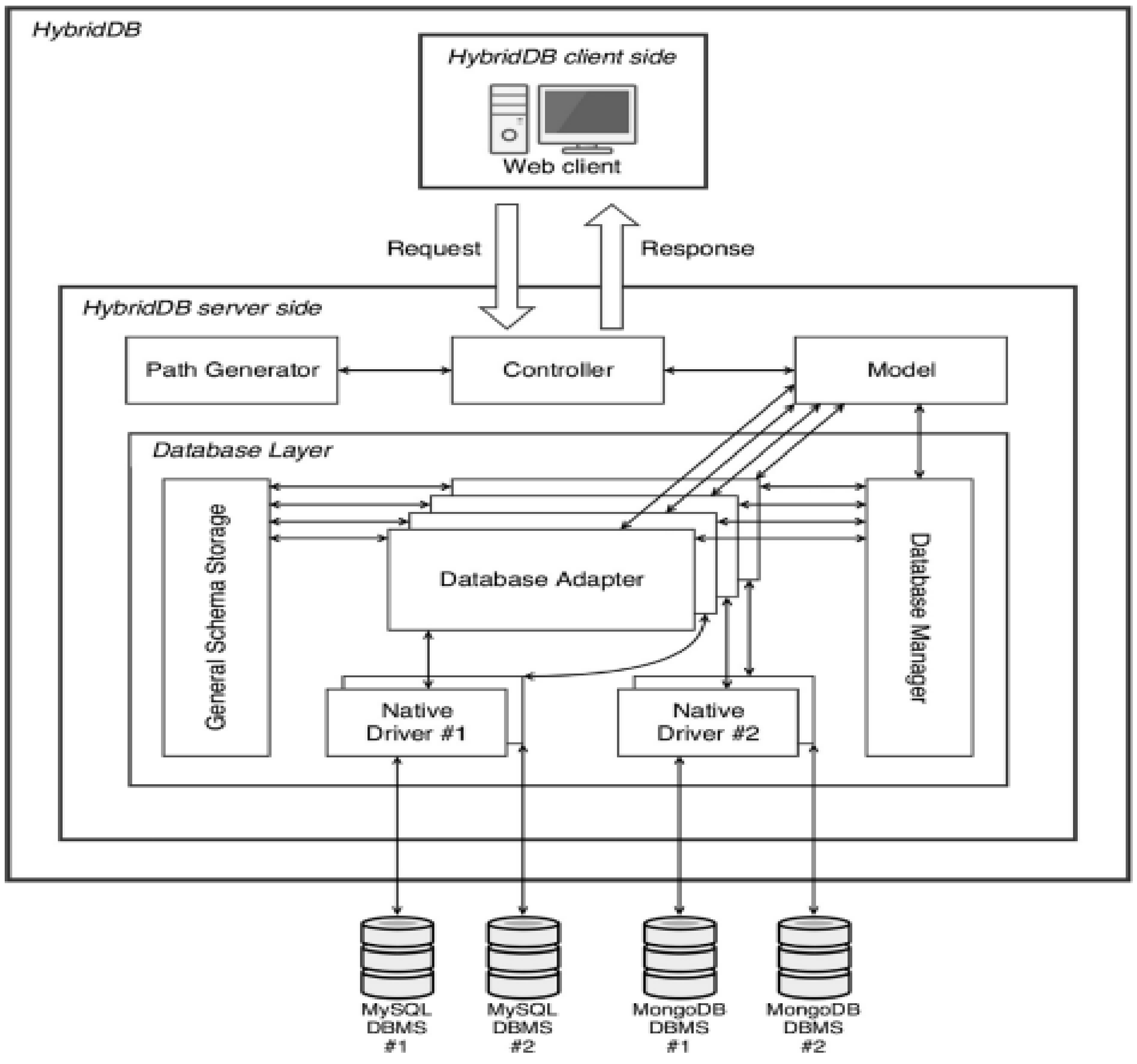

Figure 1: Illustration of the Hybrid DB

The database layer provides all the database services to the source systems. On the segment, the connection parameters are set, generate and manage the general schemas, and perform database procedures on the source databases. The database layer has modules that ensure it works seamlessly, the database manager, database adapter, general schema storage and native drivers for all supported database system types. The database manager manages database connections.

Therefore, during the linking of a new database the database manager records the parameters of the new database connection. However, databases can be deleted, and parameters of existing database can also be altered. The general schema storage records the general schemas while the database adapters record their descriptions. The primary role of the database adapter is to connect with the correct native driver and make database queries on the source databases. The hybrid database has a path generator, controller and model. The path generator handles the generation of alternative paths for queries that require more than one relation or collection. The controller governs the implementation of business logic and connects to the web client which means it is responsible for requests and responses to and from the browser. Additionally, the module authenticates all parameters from the client and uses the model to call database queries on the adapters. In terms of performance, the running times of the hybrid database are mainly reliant on the speed of native database queries and the transfer speed of data from the source database to the hybrid database.

SQL++

SQL++ is a merging semi-structured data model and query language that was developed to incorporate the data model and query language functionalities of NoSQL, NewSQL and SQL on-Hadoop databases. Its semantics are founded on the extensive work done by the database research and development community in non-relational data models and querying languages, OQL, and XQuery. Object Query Language (OQL) is a SQL-like query language with a nested relational model, while XML Query (XQuery) is a query and functional programming language that detects and extracts elements and attributes from XML documents. SQL++ is the right solution since most databases do not support all the functionalities of the standard SQL database, and semi-structured data models and query languages encompass SQL's features.

Conclusion

This overall purpose of the thesis is to analyze the challenges involved in the extraction of data from NoSQL databases and suggest potential solutions. Several possible solutions were analyzed, such as Cassandra and Neo4j tools with Spotfire, the unified data access platform, and SQL++. Each method had its unique strengths and weaknesses. The major challenge of NoSQL systems is that they have diverse data models, no explicit schemas and the lack of a general query interface. The suggested solutions and tools discussed in the paper demonstrate ways in which data extraction and other NoSQL challenges can be solved. For instance, Neo4j and Cassandra tools require the identification of the data structure in the NoSQL database that stores the source data which is an assortment of entity instances. In terms of model, the instances can be easily mapped to Spotfire's tabular data model. Schemas for the structures can be determined through the sampling of the database, which is followed by clustering. On the other hand, data extraction requires a comprehensive analysis of the source databases particularly its query language, and sections of it can be given to the user to allow adequate data extraction.

On the other hand, uniform data access relies on data integration, which raised the issue of heterogeneity of database systems. The platform is merging data, thus enabling data querying from diverse databases at the same time. In the background, metadata of source database is transformed into a general schema that is represented in JSON objects. The schemas represent the syntactical and semantic variations of source databases. Every source database has a connectivity graph that is produced to determine the architecture of the database objects in the source system. It is possible to generate the linkage information of the data using the connectivity graph by using a pathfinding algorithm. Database queries are transformed into general parameter description which can be used to generate and perform DBMS queries automatically. The output from the queries are translated to JSON objects, and eventually merged by joining separate JSON files. The output can be viewed in several data visualization forms, such as tables or CSV files (Leclercq, 2016).

Work Cited

Ahlberg, Christopher. “Spotfire: an information exploration environment.” ACM SIGMOD Record 25.4 (1996): 25-29.

Blasgen, Michael W. “Database systems.” Science 215.4534 (1982): 869-872.

Fowler, Brad, Joy Godin, and Margaret Geddy. “Teaching Case Introduction to NoSQL in a Traditional Database Course.” Journal of Information Systems Education 27.2 (2016): 99-103.

Jurie, Jay D. “Structured Query Language: An Instructional Tool for Public Administration.” Public Productivity & Management Review (1992): 371-380.

Leclercq, E., et al. “Snfreezer: a platform for harvesting and storing tweets in a big data context.” Frame A. Mercier A. Brachotte G. Thimm C. eds. Tweets from the campaign trail, researching candidates’ of Twitter during the European parliamentary elections (2016): 19-33.

McCreary, Dan, and Ann Kelly. “Making sense of NoSQL.” Shelter Island: Manning (2014): 19-20.

Mohammed, Alshafie Gafaar Mhmoud, and Saife Eldin Fatoh Osman. “SQL vs. NoSQL.”

Nasholm, Peter. “Extracting data from NoSQL databases.” The University of Gothenburg, Gothunburg (2012).

Ong, Kian Win, Papakonstantinou, and Romain Vernoux. “The SQL++ unifying semi-structured query language, and an expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewsSQL databases.” CoRR, abs/1405.3631(2014).

Sahiet, Dumin, and P. D. Asanka “ETL framework design for NoSQL databases in Dataware housing." Int. J. Res. Comput. Appl. Rob 3 (2015): 67-75.

Scholz Tobias M. Big data in organizations and the role of human resource management: A complex system theory-based conceptualization. Frankfurt a. M.: Peter Lang International Academic Publishers, 2017. 9-82.

Van de Camp, M. M., Martin Reynaert, and N. H. J. Oostdijik. "WhiteLab 2.0. A web interface for corpus exploitation." (2017).

Vathy-Fogarassy, Agnes, and Tamas Hugyak. "Uniform data access platform for SQL and NoSQL database systems." Information Systems 69 (2017): 93-96.

Vatika, Sharma, and Dave Meenu. "SQL and NoSQL databases." International Journal of Advanced Research in Computer Science and Software Engineering 2.8 (2012): 20-27.

Wang, Min, Bala Iyer, and Jeffrey Scott Vitter. "Scalable mining for classification rules in relational databases." A Festschrift for Herman Rubin. Institute of Mathematical Statistics, 2004. 348-377.

Zou, Beibei, et al. "Data mining using relational database management systems." Pacific-Asia conference on knowledge discovery and data mining. Springer, Berlin, Heidelberg, 2006.