Hybrid Databases

The next generation of data management

Sneha Shankar
Department of Computer Science
University of California, Los Angeles
Los Angeles, USA
snehashankar@cs.ucla.edu

***Abstract*—The growing importance of Big Data in today's world has increased the popularity of NoSQL databases. These non-relational data-stores have now become an important type of database management systems because of their scalability power. However, this does not mean the popularity of traditional, well established SQL databases have dwindled. With an aim of getting the best capabilities of both these database management systems, this survey paper throws light on the use of Hybrid Databases. They help to bridge the gap between SQL and NoSQL databases. Generally, data is stored in the suitable type of database in order to reap maximum benefits. This is observed when a part of data is unstructured and is fit to be stored in a NoSQL database while the rest of it is perfectly relational and should be stored in a relational database. This division of data across more than one data-store creates a gap in terms of uniform access of data. In order to bridge this gap, we describe two approaches of using a Hybrid Database by creating an abstraction layer on top of the data-stores. The first type of abstraction layer encompasses a mechanism to load NoSQL data in a SQL database in triple format and thereby develop a query language to retrieve this non-relational data from a relational database. The second method highlights a generalizable SQL interface called as the Integration and Virtualization Engine which can be used as an abstraction layer to have SQL access to a NoSQL database. The architectures of both these approaches are also briefly explained in this paper. We also comment on the advantages, practical importance and implementation of both these methods given the user constraints and requirements.**

***Index Terms*—SQL, NoSQL, MongoDB, triples, abstraction, integration, virtualization.**

I. Introduction

Relational Database Management Systems (RDBMS) are well established in the arena of data management and have well defined standards. Not only this, they have evolved with lot of research been done and have been optimized for decades. All relational databases use the Structured Query Language (SQL) for data management. These databases have dominated the enterprise domain. However, with the advent of Big Data, NoSQL (Not only SQL) databases have stolen the limelight. They are now known to support applications which the relational databases fail to support. Having said that, the trend today is not in using only a NoSQL database, but to harness the power of both relational and NoSQL databases in one database system. The next generation of data management calls for conveniently using both these database systems together in one enterprise application.

SQL databases are perfectly relational. Here, the 'relation' is defined in terms of the foreign key which can be used to reference some other attribute of a different relation. Such kind of a relation is not found in a NoSQL database. And hence the term 'Not only SQL' is actually a misnomer. While SQL databases have a well-defined schema, NoSQL databases do not have one. They are generally classified as key-value stores, document stores, graph databases and column based stores. Each of these exhibit their own unique characteristics. One of the most important feature of NoSQL databases are that they are horizontally scalable. This means such databases can accommodate variable and very heavy workloads by hosting data across multiple databases. They can add up more servers to cater to the load of the database system. Relational databases on the other hand, can at most be horizontally scalable. This refers to a condition wherein more physical resources like memory, CPU, etc. are added to the existing server. However, the cost and maintenance behind this is tremendous. That is why NoSQL databases are preferred over SQL databases for managing Big Data. Although NoSQL databases are trending today and are using cutting edge technologies, they suffer a huge drawback - they are not generalizable and do not have one particular standard like that of the traditional SQL databases. In fact, different NoSQL databases have different query languages. Which is why it becomes difficult to extend NoSQL databases to various enterprise applications. In such a case the enterprise application will have to have different supporting technologies for each of these databases. This is in contrast to that of the SQL databases where all of them can be made to interact with other applications through a JDBC/ODBC driver.

II. Hybrid Databases: Bridging SQL and NoSQL

Keeping in mind the advantages of SQL and NoSQL databases as described in Section 1, we aim to get the best of both worlds. The need for this basically arises when an application needs data storage wherein a part of data is highly unstructured and cannot be stored in a relational database, while the rest of it is perfectly relational. Let's take an example of an ecommerce [3] company which has product based data as well as CRM (Customer relationship management) data. Each

product's attributes is different from that of the other. They have different suppliers too. It is also highly possible that the product gets revised periodically. This means that though the product remains the same, the item on sale which is described by the product might change. As a result of which, such product related data should be stored in a schema-less NoSQL database. On the other hand, the CRM data held by the company stores information about the orders, suppliers, customers, payments and invoices. Since this data is relational, it is best to store it in a relational database. Thus, we see that in order to store each type of data in an appropriate way, we should ensure we have both NoSQL and SQL data-storage systems for this application. Only then, data management can prove to be worthwhile. However, there is a tradeoff associated with this. Expertise in both SQL and NoSQL languages would be required. Additional programming will be needed to combine the data from each of the source and return the correct result. The application complexity will increase and maintainability will decrease. The fact that NoSQL databases do not have any standard as of today further complicates the scenario. It is this gap of non-uniform data access that we wish to resolve. Hybrid Databases will help us bridge this gap between SQL and NoSQL databases while retaining the benefits of both the database systems. However, the solution is not trivial.

We can think of a hybrid database as an abstraction layer which can be placed on top of the databases which we wish to bridge or connect. This abstraction layer should be independent of the database systems underneath as well as the programming language used by the application. The input to this abstraction layer should be one query which returns a single result. The abstraction layer is then expected to analyze the query, identify the correct database from which to fetch data, retrieve and combine the data and lastly return the combined data fragments as a single result-set to the user. These operations are similar to that of any other database system. Hence, this type of abstraction layer, which has its own query language, can itself be classified as a database. We will call this as a Hybrid Database.

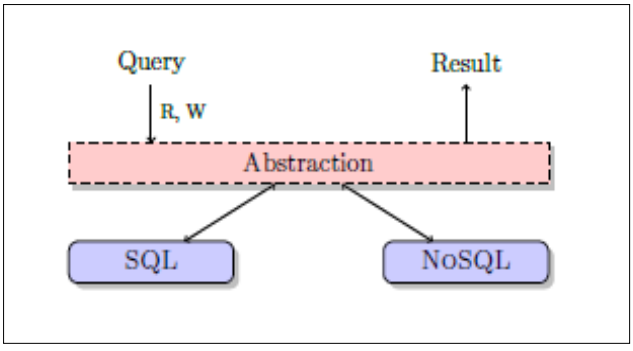

Fig. 1. Desired abstraction layer [1]

This kind of an abstraction layer can be created in different ways. Following are the three methods of implementing the same:

1. Separate Software Layer
   This encompasses programming a software application such that it acts as an independent database and performs the abstraction mechanism.

2. Load NoSQL in SQL
   This approach would signify that we use a relational database as the primary database and transform NoSQL data into a relational, well-structured data to be loaded in the relational database. This can then be retrieved using the widely used structured query language.

3. SQL access to NoSQL
   This approach implies the development of a generalizable SQL query interface, which can be used to query both relational and NoSQL databases. It is essentially a middleware which can parse the SQL queries and convert them into a query language used by the underlying NoSQL database. However, since NoSQL has limited set of functionalities as compared to SQL, one can only ensure a subset of SQL to be offered. This technique, however will be useful for data comparison and migration as well as it will allow portability.

In this survey paper, we will stress on the last two approaches mentioned above to build a Hybrid Database. We will touch upon the architectural details of these approaches and will highlight the practical importance of the two.

III. Related Work

The loading of SQL in NoSQL so as to create a Hybrid Database as described in this survey paper was inspired by [1]. The idea of having SQL access to NoSQL is described in [2]. There has also been prior research on standardizing APIs so as to use both SQL and NoSQL systems together. A common programming API which allows querying of Redis, MondoDB and HBase is described in [9]. In fact, a SQL engine was even integrated on top of HBase in [10]. This used the Apache Derby query engine to process joins and other operators not supported by HBase. In [11], a SQL interface over SimpleSQL was defined. [12] provides a generic SQL interface over key-value stores. Also, other attempts to provide a uniform way to access NoSQL systems via XQuery [8], a new query language called Unstructured Query Language (UnQL) [13] or a Java interface [14] have already been made. Mapping SQL to various query languages has also been an aspect of federated databases [15]. Another implementation of having SQL access to NoSQL systems is described in [5].

IV. NoSQL in SQL

In this approach, we will show a mechanism to represent the unstructured NoSQL data in a well-structured manner and load it in a relational SQL database as described in [1]. This will allow users to read NoSQL data using SQL queries. An obvious question which can be posed here is - why should a SQL

database be considered as the primary data storage system? There are many reasons to do so. The first and the most important reason is because relational databases and the structured query language itself are well established and standardized. All relational databases use the Structured Query Language (though with minor differences) for data management. SQL also has a very widespread and well-versed existing knowledge-base of developers. Moreover, since the advent of relational databases, they have been constantly optimized. Because of their stability, good performance and standardization, relational databases are known to be mature in the arena of data management. Extensibility and ease of adoption is another basic feature of such databases. Today, a SQL database can be connected to any external application just by using a JDBC/ODBC driver. These aspects of a SQL database make it a good candidate to serve as the main database system in a Hybrid Database. By storing NoSQL data in a relational database during query time, we will ensure that the overall number of queries to read the NoSQL data is actually reduced.

A. Triple Representation

In this part, we will see how to represent NoSQL data in a relational form. For this, we use a triple representation. As discussed in Section 1, we know that NoSQL databases do not have a generalized standard. Nor do they all adopt any one particular query language. Every NoSQL database system has its own query language. This not only creates challenges for developers and analysts working on Big Data, but also gives no interoperability even amongst various NoSQL systems. In an attempt to standardize the representation of such NoSQL data, we use a triple notation. This kind of a notation which is inspired by RDF [7] can be used to represent any arbitrary data. RDF is standardized and is a W3C recommendation.

In this representation, one or more triples can be used to represent a given data. A given triple (s, p, o) signifies that the object s is related to the object o using the predicate p. This means s will have a value of o for a data attribute p. By this definition, we can slightly transform the triple notation to make it intuitively more significant. Essentially, this (s, p, o) triple can be called as (id, key, value). This means that a triple with id as mentioned in the id field will store the value of the key as mentioned in the value and key fields respectively. We will henceforth refer to this as the (i, k, v) triple. Such set of triples can then be used to represent any arbitrary data. For example, if we want to express a data specifying 'My name is Bob', we can have a triple (h, name, Bob) which represents this arbitrary information. This means that the triple whose id is h is used to express a person whose name is Bob. Thus, we see that one triple can describe only one attribute of data. If we want to express more than one attributes of data, we can use a set of more than one triples for collective representation. For instance, now we want to express Bob's age is 25 in a set of triples. For this, we add another triple to the above mentioned triple to express Bob's age. Thus, we can use {(h, name, Bob), (h, age, 25)} to represent the required information. Note that we used the same id in both the triples to signify the data (same person) whose attributes are being represented.

To load NoSQL data in SQL, we first need to verify that both NoSQL and SQL data can be represented using triple format. The Table I (a) shows two tuples in a relation and their corresponding triple representation in Table 1 (b).

TABLE I. TRIPLE REPRESENTATION OF RELATIONAL RECORDS

id	***name***	***age***
1	Alice	20
2	Bob	25

(a) Relational representation

id	***key***	***value***
i1	id	1
i1	name	Alice
i1	age	20
i2	id	2
i2	name	Bob
i2	age	25

(b) Triple Representation

Now, let's consider we have a nested document d as follows:

d = {name: Bob, age: 25, courses: {code: CS259, grades: [A, A+]}}

Here, there are two levels of nesting within the document d. The data for courses is present in a nested document within d. We observe that the grades here are stored in a list which can also be considered as a nested document {0: A, 1: A+} within courses. The 0 and 1 signify the position of the elements in the list. Table II shows the triple representation of this document d.

TABLE II. TRIPLE REPRESENTATION OF NESTED DOCUMENT

id	***key***	***value***
i1	name	Bob
i1	age	25
i1	courses	i2
i2	code	CS259
i2	grades	i3
i3	0	A
i3	1	A+

In Table II, we observe that there are three ids (i1, i2 and i3) which are used to represent the data in document d. This is because the main structure of the document d contains two sub-structures (nested documents) within the parent structure/document. Each substructure has a unique id which should be different from the id of the parent structure. Here, we see that i2 and i3 are used to represent the substructures within the main structure of document d. We use the id of the substructure to link to its parent structure by denoting it as a value of a field which contains a nested document. In this way, we can ensure that all sets of triples remain connected and they

collectively represent the entire document. As such, we can conclude that the ids present in a triple are only used to connect the triples and are not part of the actual data.

Thus, from Table I and Table II, we now know that both SQL and NoSQL can be represented in triple format. It is easily verifiable now that this kind of a triple representation allows maximum flexibility. The NoSQL data converted into triples can now be safely loaded as a relation F (id, key, value) in a relational database. Since the triple format is well-structured, by loading this relation F in a relational database, we aren't violating any standards of RDBMS. Now that we have the relation F in a SQL database, we can join this to other relations in the database which can be queried using a SQL query, thus making it possible to combine SQL and NoSQL data in a single query result.

B. Theoretical framework for Data Reconstruction

As seen above, to fill the relation F with transformed data, we need to convert NoSQL data into triples. In this part of the section, we describe a theoretical framework for this type of data reconstruction. To connect the triples of the parent structure to the nested or the substructure, we add a triple which describes the relation between the two using the key of the nested key-value structure. Here, the most important constraint which we should follow is that each substructure should have its unique id. Without this, the following framework cannot be applied.

A formal transformation is used to express this mathematically. In this transformation, we use two functions:

- φ_i : To transform a key-value pair
- ψ_i : Expects a set of key-value pairs

These two functions can be defined as follows:

$$\varphi_i(p) = \{(i, pk, pv)\}, \quad \text{if pv is a constant}$$
$$\{(i, pk, j)\} \cup \psi_j(pv) \quad \text{if pv is a set}$$
$$\psi_i(S) = \cup\, \varphi_i(p) \quad \text{for all p in S}$$

In the above definition for φ_i, the new variable j is used to represent the id for the nested key-value pairs. Let us consider a small nested document s and apply this transformation on it so as to obtain a triple representation.

s = {name: Bob, grades: [8, 6]}

Since the document contains a nested list within it, we can say that the above document is equivalent to

s = {name: Bob, grades: {0: 8, 1: 6}}

We now apply the transformation as shown in Figure 2 to get the triple representation.

$$\begin{aligned}
\psi_{i_1}(s) &= \bigcup_{p \in s} \phi_{i_1}(p) \\
&= \phi_{i_1}(name : \text{Bob}) \cup \phi_{i_1}(grades : \{0:8, 1:6\}) \\
&= \{(i_1, name, \text{Bob})\} \cup \phi_{i_1}(grades : \{0:8, 1:6\}) \\
&= \{(i_1, name, \text{Bob})\} \cup \{(i_1, grades, i_2)\} \cup \psi_{i_2}(\{0:8, 1:6\}) \\
&= \{(i_1, name, \text{Bob}), (i_1, grades, i_2)\} \cup \bigcup_{p \in \{0:8, 1:6\}} \phi_{i_2}(p) \\
&= \{(i_1, name, \text{Bob}), (i_1, grades, i_2)\} \cup \phi_{i_2}(0:8) \cup \phi_{i_2}(1:6) \\
&= \{(i_1, name, \text{Bob}), (i_1, grades, i_2)\} \cup \{(i_2, 0, 8)\} \cup \{(i_2, 1, 6)\} \\
&= \{(i_1, name, \text{Bob}), (i_1, grades, i_2), (i_2, 0, 8), (i_2, 1, 6)\}
\end{aligned}$$

Fig. 2. Triple reconstruction using formal translation framework [1]

Thus we see that the document is now converted into a set of 4 triples.

C. Data Retrieval

Now that we have stored the NoSQL data as a relation in a SQL database, the task is to retrieve this data. Though this kind of a triple representation is flexible, it comes with a trade-off. Since one data entity of NoSQL is stored in the form of multiple records in F, we should retrieve the original data from all of them. Basic relational algebra seems to be a plausible solution for this. Let us consider a set of triples as in Table I (b). To combine the data attributes i.e. id, name and age in this case, we perform self joins on the relation F and ensure that the id values are equal. The relations T1, T2 and T3 as shown in Tables III (a), (b) and (c) are reconstructed using the following relational algebra expressions.

T1 = $\rho_{T1\ (id)}\ (\pi\ vi\ (\sigma_{ki\ =\ id}\ (\ \rho_{Ti}\ (\ i,\ ki,\ vi)\ (T)\ )\ )\ )$

T2 = $\rho_{T2\ \ (id,name)}\ (\pi_{\ vi,vn}\ (\ \sigma_{ki\ =\ id\ \wedge\ kn=name}\ (\ \rho_{Ti\ (\ i,\ ki,\ vi)}\ (T) \bowtie \pi\ \rho_{Tn(\ i,\ kn,\ vn)}\ (T)\ )\ )\ )$

T3= $\rho_{T3\ \ (id,name,age)}\ (\pi_{\ vi,vn,va}\ (\ \sigma_{ki\ =\ id\ \wedge\ kn=name\ \wedge\ ka=age}\ (\ \rho_{Ti\ (\ i,\ ki,\ vi)}\ (T) \bowtie \rho_{Tn(\ i,\ kn,\ vn)}\ (T) \bowtie \rho_{Ta(\ i,\ ka,\ va)}\ (T)\ )\ )\ )$

The first relation is a projection of only the id from the triple relation where the key is id. In the second relation we aim to retrieve the id as well as the name from the set of triples. For this purpose, we perform a self-join on the existing relation F. Self-join essentially means that we join every row of F to its every other row. This way, we can invoke a selection operator on only the rows having the keys as id and name in the respective join order. Every row is now going to contain six fields. We then use the projection operator to retrieve only the id and name fields. This relational algebra is specified in relation T2. Likewise, if we wish to retrieve the id, name as well as age, we perform two self joins on the triple relation set and perform a selection followed by projection of the required fields as shown in relation T3. In general, if we want to retrieve n attributes of data, then n-1 self joins are required. The Tables in III (a), (b) and (c) thus show the retrieved result-set in relational form.

TABLE III. STEPWISE DATA RECONSTRUCTION AND RETRIEVAL

id
1
2

(a) T1

id	*name*
1	Alice
2	Bob

(b) T2

id	*name*	*age*
1	Alice	20
2	Bob	25

(c) T3

In this way, any relation containing triples can be retrieved using basic relational algebra. In the next part, we will see how we use a query language which employs the above relational algebra and thus gives us one result set for the query supplied to the Hybrid Database.

D. Query Language and Collaboration with SQL

In order to describe the self joins mentioned in the above section, a concise description of how the triples should be combined is necessary. The SPARQL (pronounced as sparkle) query language is used in this context. The main reason behind choosing SPARQL is that it is obeys the basic graph pattern. And such a pattern resembles a set of RDF-like triples to describe data. Since we are using RDF triples for our reconstruction, SPARQL seems to be the best fit to query and retrieve such data.

SPARQL consists of variables and the values of these variables are bound only to them. This means that if the same variable is used more than once, then the parts of data where the query is performed should also contain the same value so as to match the basic graph pattern. Let us consider the set of triples as derived in Section 4 B. Let Table IV signify the relation in an SQL database where these triples are stored.

TABLE IV. TRIPLE RELATION FOR S

id	*key*	*value*
i1	name	Bob
i1	grades	i2
i2	0	8
i2	1	6

This triple relation can then be described using the basic graph pattern as below:

```
?i name ?name .
?i grades ?j .
?j 0 ?f .
?j 1 ?s .
```

This can now be converted into a style resembling the set of triples. The modified result is shown below:

```
(?i, name, ?name),
(?i, grades, ?j),
(?j, 0, ?f),
(?j, 1, ?s)
```

We know that the variables i and j are used only to connect the triples together. Therefore, we can change their notation and use them as references in the basic graph pattern. Such references are called as navigational variables. The modified notation can be represented as follows:

```
(#i, name, ?name),
(#i, grades, #j),
(#j, 0, ?f),
(#j, 1, ?s)
```

Now that we have derived this representation, we can formulate a NoSQL query pattern from this. We combine the different NoSQL query pattern triples into a nested key-value set if they have the same navigational variable as their id value. This NoSQL query pattern can be represented as follows:

```
(
   name: ?name,
   grades: (
      0: ?f,
      1: ?s,
   )
)
```

We can now embed this query pattern into our SQL query and thereby retrieve data collectively. This collaboration of NoSQL query pattern with a SQL query is as follows:

```
SELECT
   r.name
FROM
   NoSQL(
      name: ?name,
      grades: (
         0: ?f,
         1: ?s,
      )
   ) AS r
WHERE
   r.f = 8
```

The above query retrieves the name of the person who has the first grade as 8. Such kind of a query pattern allows easy transformation of NoSQL to SQL and also provides isolation of the NoSQL query.

E. Architectural Overview

Figure 3 represents the architecture of the framework described in Section IV. The yellow region in the figure represents the translation from NoSQL query pattern to an equivalent SQL fragment. The green region depicts the construction of triples from NoSQL data. This set of triples is then streamed in to an SQL database during query time. The blue region specifies the

actual execution on the SQL data after merging queries to be executed both on actual SQL and actual NoSQL data. However, this architecture has some drawbacks.

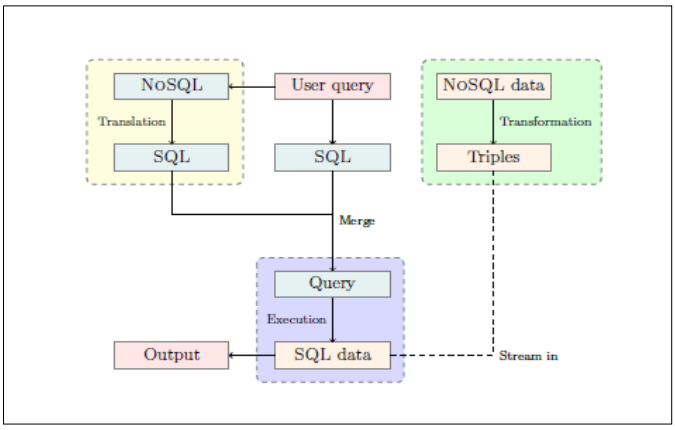

Fig. 3. Architectural workflow of the life of a query [1]

F. Drawbacks

Firstly, the translation from NoSQL query pattern to an equivalent SQL query should be performed automatically prior to query execution. Secondly, there is a high possibility that the NoSQL database might not be available when data is to be streamed in to the SQL database. This might result in an empty relation F. This phenomenon is called as query atomicity violation. Thirdly, a user might modify the NoSQL data during streaming which will thereby result into an inconsistent query result. One solution to avoid these problems is to stream the data into the SQL database before query execution. However, this is next to impossible as we do not know which data is to be loaded in the relational database and how much data will the database be capable of storing. It is very silly to think of loading all of the NoSQL data into the SQL database which destroys the entire essence of a NoSQL database itself. To overcome these difficulties, one can use the approach described in Section 5 to create a Hybrid Database.

V. SQL ACCESS TO NOSQL

NoSQL database systems are capable of handling large volumes of data because of their ability to scale. They are often open source and adopt cutting edge technologies. However, till present, there is not a common way to interface NoSQL systems and thereby there is no generalization. As such, a standardized API will take us one step further to get generalized NoSQL systems. This API should be such that it will make it easier to switch between different NoSQL systems with no additional programming expertise. In this section, we will introduce one such common interface to bridge different data systems such that it becomes easier to compare the data in them, enhance portability and thus enable migration of applications between them.

Inspired by [2], we throw light into the design and creation of a generalizable SQL interface for both relational and NoSQL systems. This will ensure that we retain the benefits of SQL in the context of NoSQL database systems without compromising the benefits of the underlying NoSQL system. We introduce a middleware called as Unity which will act as this generalizable SQL interface. Unity will take input of SQL query and will allow it to be automatically parsed, translated and executed using the underlying API of the respective data sources. Unity is an integration and virtualization system and is capable of even performing joins across systems. It is this feature of Unity which allows combining the data retrieved from more than one systems. In this way NoSQL data sources can be queried using SQL. However, due to the limited functionality of NoSQL systems, only a subset of SQL can be offered. The primary motivational factors to support such kind of SQL access to NoSQL systems are as follows:

- SQL is a declarative language. This means it allows queries to be descriptive while at the same time hiding the implementation and query execution details
- SQL is a well-established and standardized language which allows portability between systems
- SQL already has a massive existing knowledge base of database developers
- SQL access to NoSQL will allow NoSQL databases to interact with enterprise systems to which SQL can connect to with JDBC/ODBC drivers

In the following parts of this section, we will explain how the Unity is built and how does it execute a SQL query in two different databases. We show the latter by delineating a query execution plan. Lastly, we also touch base upon the experiment conducted to gauge its performance.

A. Unity Architecture

In this section, we briefly describe the architecture of Unity Integration and Virtualization Engine. The main components of this architecture are the SQL Query Parser, Query Translator, Optimizer and the Virtualization Execution Engine. Additionally, there is also support for Schema Generator as well as for the Function and Dialect Translator.

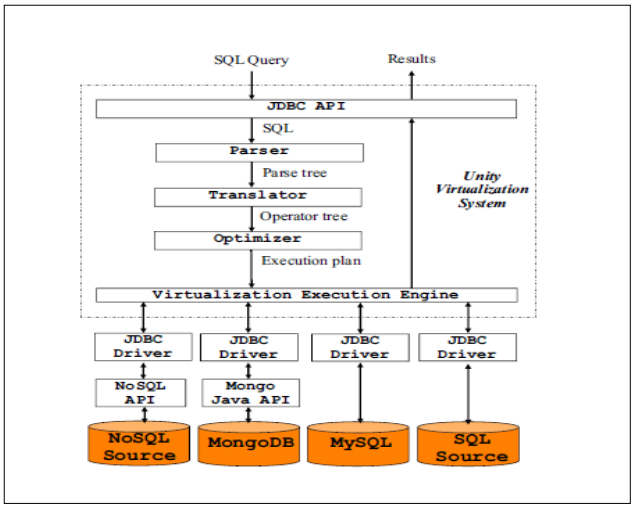

Fig. 4. Unity Architecture [2]

a) SQL Query Parser

The SQL Query Parser takes a SQL query as input and converts it into a parse tree. At the same time, it also performs validation on the query. If the validation fails, it will throw an exception to the user. In Unity, the query parser is implemented using JavaCC. It supports standard SQL-92 syntax for SELECT, INSERT, DELETE and UPDATE. It also supports statements including inner and outer joins, subqueries, GROUP BY, ORDER BY and HAVING. After the validation, the converted parse tree represents the input SQL query.

b) Query Translator

The query translator takes the parse tree supplied by the query parser and translates it into a relational operator tree. For the query is to be executed on a SQL database, it performs validation on the table names and fields provided against the relational schema of the underlying database. For the part of the query to be executed on a NoSQL database, however, there is no validation and the translator passes the query directly to the execution engine.

c) Schema Generator

We know that NoSQL systems such as MongoDB do not have a defined schema to store and manage data. If Unity is interfaced with MongoDB, it has the capability to optionally generate a schema for the NoSQL data. A schema for MongoDB is created by sampling the data in each of its collection. That is, if different items in the collection have the fields (a, b), (a, d, e) and (b, d, f), then the relational schema will be (a, b, c, d, e, f). Once created, this schema can be optionally stored in a MongoDB collection itself.

d) Query Optimizer

The aim of a query optimizer is to push down as much of the query on the individual data sources so as to receive better performance. If the query involves a single relational source, then the entire query plan will be executed on that source. However, for NoSQL systems, an SQL query may not be completely executable in it. This is because NoSQL generally does not provide the functionality of grouping/aggregation, joins and subqueries. Therefore, while writing an SQL query, a user should be careful to not invoke these on a NoSQL system. For queries involving multiple data sources, the optimizer first separates out the relational operators generated by the translator. It then identifies which operator should be used on which source. After this, it pushes down as much of the computation to each source as possible. Now, the optimizer needs to join the results obtained from different sources. To do this join, it first plans what will be the optimal ordering so as to combine the results and return a single query result-set to the user. The various techniques used by the query optimizer are as follows:

- Push-down filters: These filters push down the selection operator on the respective data sources for execution
- Join ordering: This is determined using a cost-based optimizer. The optimizer uses costs that are higher when the join is executed across different systems rather than within a particular system
- Push-down, staged joins: A staged join is used to combine the results across more than one systems. It will retrieve the results from one join result and use it to modify the part of the query to be sent to the other source. The data from each of the source is extracted in parallel using separate threads and then joined in the virtualization engine using a hash join. This type of join is very powerful. A staged join will be very efficient if the number of rows coming back from one source is much smaller than the other source. These rows are then used to modify the query which is sent to the other source. This source will then perform the join it executes its own query. The results returned from this source will be only those rows in the join result. After this, the virtualization engine will perform a merge operation on any attributes from the first source query back into the returned rows from the second source.

As such, the optimizer tries to maximize the size of a query plan sent to each source. However, the operator cannot be sent to a source if it is not supported by that system. Such operations are executed by the virtualization system.

e) Execution Engine

The engine is the entity which will perform the actual execution. The results are retrieved in a JDBC ResultSet. However, since NoSQL systems do not return a JDBC ResultSet, a JDBC wrapper is placed above the NoSQL data-source's API which will accept a SQL query and use the engine to parse and convert it.

f) MongoDB SQL/JDBC

We know that MongoDB does not understand SQL. We need to communicate with MongoDB only with its own query language. For this purpose, we use a JDBC driver which will accept an SQL query. This driver will also have a parser and converter which will generate a parse tree and from that a relational operator tree respectively. The operators in this tree will then be mapped to the corresponding operator of a MongoDB API call. It will throw an exception if the conversion is not possible.

g) Function and Dialect Translator

Though all relational sources use the SQL query language, there are some minute differences in some of the functions though the functionality essentially remains the same. Thus, the role of a function and dialect translator is to create a mapping for each data source's function and SQL dialect feature which explains how it is supported for a database. If there is no mapping, then it means the syntax is not supported; which will result in an exception being thrown.

B. Execution

We now explain the execution of a query plan in the Unity engine. Let's consider an example where two data sources are used in an application. Let the customer details be stored in a customer database (NoSQL) and the order details in an order database (SQL).

Customer (cid, cname, addr.street, addr.city)
Orders (oid, cid, odate, total)

Let's say the NoSQL database is MongoDB and it stores a sample customer object in its collection as follows:
{"cid":1, "cname":"Fred", "address": {"street": "Main", "city": "New York"}}

Let a sample record stored in a table of the Orders database be:
(100, 1, '2013-11-10', 35.45)

Now, let's say we want to retrieve the total of all orders for the customer 'Fred' since January 1st, 2013. Then, we run the following query:

```
SELECT SUM(total) as totalAmount
FROM mongo.customer C
INNER JOIN mysql.orders O ON C.cid=O.cid
WHERE C.cname = 'Fred
AND O.odate > '2013-01-01'
```

The SQL queries executed by each of the source as depicted above are:
MongoDB:

```
SELECT cid FROM customer
WHERE cname = 'Fred'
```

MongoDB JSON query format:
db.customer.find({"cname":"Fred"}, {"cid":1})

MySQL:

```
SELECT cid, total
FROM orders
WHERE odate > '2013-01-01' and cid IN (1)
```

The above query is optimized into the execution plan as in the Figure 5.

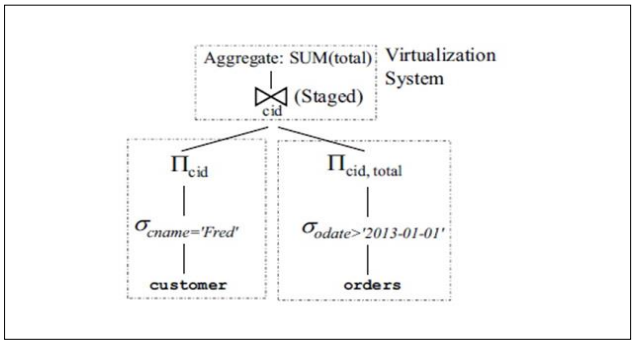

Fig. 5. Query Execution Plan [2]

C. Experimental Results

Unity's performance was measured first on MongoDB only and then by letting it interface a MySQL database and MongoDB. In the former case, the SQL to NoSQL transition was tested using an orders database comprising of 15000 records. Unity's performance was compared with that of MongoDB's performance without the abstraction layer. This involved executing 10,000 operations each for SELECT, INSERT, DELETE and UPDATE. These results are tabulated in Table V and are the average of 3 runs.

TABLE V. QUERY EXECUTION PERFORMANCE (IN SECONDS)

	Mongo	***Unity***	***% Diff***
SELECT (key)	99.4	101.1	2
SELECT (scan)	25.1	28.0	12
INSERT	11.1	14.3	30
DELETE	109.8	111.7	2
UPDATE	109.5	123.1	12

The Diff column in the Table V states the overhead percentage of Unity as compared to MongoDB. We observe that for all other operations apart from INSERT, the overhead is quite less i.e. less than 15%. This is tolerable and in fact seems very trivial in front of the advantages offered by the generalized SQL interface. The overhead for INSERT in this case is high because the insertion time is so low that the overhead is actually more of a factor. This overhead is fixed-cost independent of the query size and hence the relative overhead will always decrease as the query time increases. Which means that the insertion performance will be much better with larger data sets having more users.

Unity was also tested on two databases: MySQL and MongoDB. The join combined a customer table with 1500 records with an orders table containing 15,000 records. The observations found were: time to read the base relations from Mongo is 0.9 seconds and that from MySQL is 1.5 seconds. The time taken by MySQL to produce the join result is 2.7 seconds. Unity performs the join when both tables come from Mongo in 1.1 seconds and where one table comes from Mongo and the other from MySQL in 1.7 seconds. Since MongoDB cannot perform joins, the virtualization engine provides an efficient way to join the collections in MongoDB.

VI. CONCLUSION AND FUTURE WORK

Unity or Unity-like integration and virtualization systems are used nowadays in the industry which allow SQL queries to be executed over both relational and NoSQL systems. They exhibit more benefits as compared to loading NoSQL in SQL. This is primarily because in systems like Unity, we retain the benefits of NoSQL databases while providing the entire system benefits of the SQL interface as well without any extra non-trivial cost. The virtualization layer allows translating SQL queries to NoSQL APIs as well as automatically executes operations like joins which are not allowed by NoSQL systems. This also ensures great interoperability among different database systems. Moreover it allows NoSQL databases to seamlessly interact with other enterprise applications because of the SQL interface. The Unity system has been commercially released as UnityJDBC and the translator for MongoDB packaged as a JDBC driver.

Though a lot of work has already been done in Unity, it still supports only the MongoDB NoSQL database. Future work in this arena involves setting a benchmark performance of other NoSQL systems like Cassandra. There is also work in progress to parallelize this for a cluster environment.

REFERENCES

[1] Roijackers, John, and G. Fletcher. "Bridging sql and nosql." Master's thesis, Eindhoven University of Technology (2012).

[2] Lawrence, Ramon. "Integration and virtualization of relational SQL and NoSQL systems including MySQL and MongoDB." Computational Science and Computational Intelligence (CSCI), 2014 International Conference on. Vol. 1. IEEE, 2014

[3] Rakesh Agrawal, Amit Somani, and Yirong Xu. “Storage and querying of e-commerce data”. In Proceedings of the 27th VLDB Conference, VLDB ’01. VLDB Endowment, 2001.

[4] Sharma, Vatika, and Meenu Dave. "Sql and nosql databases." International Journal of Advanced Research in Computer Science and Software Engineering 2.8 (2012).

[5] Rith, Julian, Philipp S. Lehmayr, and Klaus Meyer-Wegener. "Speaking in tongues: SQL access to NoSQLsystems." Proceedings of the 29th Annual ACM Symposium on Applied Computing. ACM, 2014.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. “An efficient SQL-based RDF querying scheme”. In Proceedings of the 31st VLDB Conference, VLDB ’05, pages 1216–1227. VLDB Endowment, 2005.

[8] H. Valer, C. Sauer, and T. Harder. “XQuery processing over NoSQL stores”. In Proceedings of the 25th GI-Workshop "Grundlagen von Datenbanken 2013", Ilmenau, Germany, May 28 - 31, 2013 2013, 2012.

[9] P. Atzeni, F. Bugiotti, and L. Rossi, “Uniform Access to Non-relational Database Systems: The SOS Platform,” in CAiSE, 2012, pp. 160–174.

[10] R. Vilac¸a, F. Cruz, J. Pereira, and R. Oliveira, “An Effective Scalable SQL Engine for NoSQL Databases,” in Distributed Applications and Interoperable Systems. Springer, 2013, pp. 155–168.

[11] A. Calil and R. dos Santos Mello, “SimpleSQL: A Relational Layer for SimpleDB,” in ADBIS, 2012, pp. 99–110.

[12] J. Tatemura, O. Po, W.-P. Hsiung, and H. Hacig¨um¨us, “Partiqle: an elastic SQL engine over key-value stores,” in SIGMOD Conference, 2012, pp. 629–632.

[13] UnQL. http://unql.sqlite.org/

[14] P. Atzeni, F. Bugiotti, and L. Rossi. “SOS (save our systems): a uniform programming interface for non-relational systems”. In Proc. 15th Int. Conf. on Extending Database Technology, EDBT'12, pages 582-585, New York, NY, USA, 2012. ACM.

[15] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Comput. Surv., 22(3):183-236, Sept. 1990.