Received: 31 October 2016 | Revised: 15 March 2017 | Accepted: 17 March 2017
DOI: 10.1002/nem.1976

SPECIAL ISSUE PAPER

WILEY

Data transformation as a means towards dynamic data storage and polyglot persistence

Thomas Vanhove | Merlijn Sebrechts | Gregory Van Seghbroeck | Tim Wauters | Bruno Volckaert | Filip De Turck

Department of Information Technology, Ghent University - imec, IDLab, Ghent, Belgium

Correspondence
Thomas Vanhove, Department of Information Technology, Ghent University - imec, IDLab, Technologiepark-Zwijnaarde 15, B-9052 Ghent, Belgium.
Email: thomas.vanhove@intec.ugent.be

Funding information
iMinds SEQUOIA

Summary

Legacy applications have been built around the concept of storing their data in one relational data store. However, with the current differentiation in data store technologies as a consequence of the NoSQL paradigm, new and possibly more performant storage solutions are available to all applications. The concept of dynamic storage makes sure that application data are always stored in the most optimal data store at a given time to increase application performance. Additionally, polyglot persistence aims to push this performance even further by storing each different data type of an application in the data store technology best suited for it. To get legacy applications into dynamic storage and polyglot persistence, schema and data transformations between data store technologies are needed. This usually infers application redesigns as well to support the new data stores. This paper proposes such a transformation approach through a canonical model. It is based on the Lambda architecture to ensure no application downtime is needed during the transformation process, and after the transformation, the application can continue to query in the original query language, thus requiring no application code changes.

KEYWORDS
big data, data transformation, dynamic data storage, lambda architecture, polyglot persistence

1 | INTRODUCTION

Relational data stores are an important building brick for legacy applications in their data storage strategy. However, with growing data sets in the age of big data analytics, applications' demands have exceeded the capabilities of classic relational database management systems (RDBMS). With this new paradigm for large-scale processing, fast access to the data is necessary. Many new systems have been designed aimed to scale horizontally, providing read/write operations distributed over many servers.[1] Many of these new systems can be categorized as NoSQL, which stands for "Not only SQL." Contrary to the classic relational databases, they provide easy scaling and performance advantages in specific scenarios, depending on the chosen NoSQL data store.[2] Additionally, they provide a more flexible or even schema-less data model, allowing rapid changes in the model. The popularity of these data stores can be measured by the sheer amount of solutions available. However, this does not mean relational databases do not have a role to play in the big data story.[3] An example is Google's F1 hybrid database,[4] a scalable distributed Structured Query Language (SQL) database built on top of their globally distributed and synchronously replicated database, Spanner.[5] Google uses this database to support their AdWords business, an ecosystem with hundreds of concurrent applications and thousands of users sharing the database, over 100TB in size in 2013.

The amount of possible solutions for data storage led to a specialization of these data stores to distinguish themselves from each other, making different data stores more suitable for different types of data or for the different use of data. Thus, a correct choice in data store is paramount for the optimal performance of the application. However, as applications tend to evolve with frequent updates and changing user numbers, the optimal choice of data store may change over the course of the application's lifespan. The concept of dynamic storage allows the stored data to be stored in the optimal format for the application at all times, transforming the format when necessary, ie, when certain requirements are no longer met. Along with this, applications often work with different types of data, eg, e-commerce platforms. Another interesting concept would therefore be to have the application use multiple data stores simultaneously, instead of forcing all data into one solution. This is often referred to as polyglot persistence.[3,6,7] In the case of a network monitoring platform, device information can be stored in a classic RDBMS, while logging data might be a better fit for a document data store or a search server such as ElasticSearch.

Introducing dynamic storage and/or polyglot persistence in existing legacy applications requires a transformation of existing data stores or parts thereof. On the one hand, there is the cost of transforming the data format, but on the other hand, many application changes may be necessary as well to support the new format. Additionally, with applications having to meet specific service-level agreements, this migration and/or transformation has to occur with as limited downtime as possible, preferably eliminating the downtime entirely in a best case scenario. This high migration and transformation cost discourages application developers to change data stores in live applications.

Based on the previous paragraphs, 3 main obstacles can be defined that currently hamper dynamic storage and polyglot persistence:

1. Migration of data to the cloud (or between clouds)
2. Transformation of data formats
3. Alteration of application code

This paper reports on advances that have been made into overcoming these obstacles as well as contributing to a new approach of data transformation in such a way that the downtime of the applications is eliminated, without additional development and implementation effort.[8] The proposed solution aims to tackle the issues concerning polyglot persistence, ie, enable applications to access and store data in different data stores simultaneously and allow for dynamic changeovers between supported data stores based on monitoring information or the intervention of the customer or administrator. The proposed solution makes use of a new open source Platform-as-a-Service (PaaS) called Tengu.[9] The Tengu platform provides researchers a time-saving approach for building big data analysis frameworks through automated installation, configuration, and integration of big data analysis and storage technologies.[9,10]

The remainder of this paper is structured as follows: Section 2 reviews related work in the different domains that already contribute to the solution of the previously stated obstacles. The approach and general workflow of the transformation are discussed in Section 3, while Section 4 describes the transformation algorithm. In Section 5, the implementation of the algorithm and workflow on the Tengu platform are detailed. Section 6 discusses the evaluation of the implementation through performance testing. Finally, the paper is concluded in Section 7 and offers several leads towards future work.

2 | RELATED WORK

Early work on data transformation[11,12] led into what are now called extract-transform-load (ETL) processes. These software processes are commonly used in data warehouses where they extract data from often different data sources, transform the data in the correct format, and load the transformed data into the data warehouse. Research in ETL has focused on modeling, efficiency, and facilitation of construction.[12] While the approach and algorithm described in this paper show several similarities to ETL processes, they are vastly different. Extract-transform-load facilitates data transformation between 2 data points, data source and data warehouse, where both data schemas are known. If a change is made in the data schema of the data source or the data warehouse, changes will need to be made in the ETL process. The proposed transformation in this paper works between 2 data points where only one data schema is known, the source data store, which is then transformed into a data schema representation of the new data store.

For each of the ETL subprocesses (extract, transform, and load), a research domain exists. Extract and load have been heavily researched as part of data migration and has become even more apparent with the complexity introduced by clouds and big data.[13] Data migration obstacles have been solved in several ways using high-performance networks,[13] workload-aware strategies,[14] and cost-minimizing approaches.[15] In this research domain, several solutions have also been proposed for live data migration, ie, a migration where a live application needs to be supported without downtime.[16,17] Other migration tools allow data from an RDBMS to be analyzed by big data processing tools, such as Hadoop. Apache Sqoop provides a framework to transfer data between an RDBMS and Hadoop.[18] The RDBMS data can thus be used in a big data analysis process after which the results can be migrated back to the RDBMS.

Another important research domain related to data transformation is that of schema matching and mapping.[19,20] It is

TABLE 1 State of the art in the domain of migration, transformation, and alteration of application code as used at Amazon, Google, and Microsoft

	Amazon AWS	Google Cloud	Microsoft Azure
Migration (online)	Database Migration Service	Storage Transfer Service	Azure Migration Wizard
Migration (offline)	Amazon Snowball - AWS Import/Export Disk	*Third party support*	Import/Export service
Transformation	Schema Conversion Tool	X	*Limited*
Alteration	Schema Conversion Tool	X	X

a process that identifies if 2 data schemas are semantically similar and describes the transformations for data to be represented in the other schema. This research domain is closely related to ETL, as it aids in the creation of the transformation subprocess. This work leverages the input of 2 data schema and maps the transformation between them, contrary to the work in this paper. Other work in the transformation research focuses on data transformation, more specifically between SQL and NoSQL data stores. However, compared to the work in this paper, it is often limited in the support of data store technologies (eg, only supporting column data stores).[21,22] The transformation approach and algorithm in the paper is aimed towards flexibility and extensibility, in theory able to support any data store technology.

Finally, direct transformation tools between 2 specific data store technologies exist as well, such as Mongify for SQL to MongoDB.[23] They are able to transform data stores from one specific data technology to another. Compared to the approach in this paper, these tools are limited, as they only provide between 2 specific data stores with no easy way of extending the support to other data store technologies. Moreover, these tools are often built with custom code and therefore do not scale well when working with legacy or production data stores in general. Another example of a direct transformation tool is present in Cassandra, using Apache Sqoop.[24] It capitalizes on the similarities between SQL and Cassandra Querying Language (CQL) to import and export data between Cassandra and a classic RDBMS. While technically transforming data between 2 different technologies, this approach provides data migration functionality. In contrast to the contributions in this paper, the Cassandra transformation tool also does not provide any optimizations in its supported technologies to decrease query latencies.

Several of the previously mentioned research topics have already seen applications in the industry. Table 1 gives an overview of how the major cloud providers, Amazon, Google, and Microsoft, overcome the obstacles described in the previous section: migration of data, transformation of data, and if alteration is required in the application code. The 'X' marks that no tool is made available by the cloud provider for a specific obstacle. At first sight, all providers supply tools for the migration of data from and towards their platform, both offline and online. Online tools allow for the migration of data over the internet while offline tools are organized processes of sending physical disks to the providers. Amazon outperforms Google and Microsoft, as it provides not only tools to overcome migration but transformation and alteration as well. However, when taking a closer look at the schema conversion tool offered by Amazon, it is mostly restricted to data stores with SQL-like querying languages for both transformation and the alteration of application code.* The tool can tweak the SQL schema of a source data store and alter the SQL query in the application code to reflect the changes made to the schema. Compared to the work in this paper, the AWS schema conversion tool is limited, as it only supports SQL-related data stores. Furthermore, this transformation still requires changes in the application, although these are executed automatically by the tool. While Google and Microsoft have no tools for a full transformation and alteration, Microsoft Azure does provide a tool for schema matching/mapping.

One of the goals of transformation in this paper is to support polyglot persistence for legacy applications. A lot of research has gone into solutions that shield the complexity of having to deal with multiple query languages through abstract data layers, such as Hibernate ORM/OGM[25] and Apache Drill.[26] These abstract data layers provide access to different data stores without the need for the application or developer to be aware of the complexities of the data store. Most of these provide only limited or no support for the migration of data between supported data stores but do allow applications to store data in different parallel data stores depending on the type of data. Many of these abstract data layers however require applications to use the abstract data layer's querying language, which in some cases is the SQL standard, but in others a custom dialect (eg, Hibernate Query Language). The abstract data layers effectively shield the data store complexity of polyglot persistence, but only for new applications. Legacy applications with big data sets still have no out-of-the-box solution to help them benefit from these new paradigms.

3 | DATA TRANSFORMATION FRAMEWORK

This section describes the approach and workflow of the data transformation as a means to achieve dynamic storage and

*https://aws.amazon.com/dms

polyglot persistence for applications. First, an architecture for the transformation is proposed that overcomes the aforementioned obstacles and avoids application downtime. Next, the approach of the actual transformation is discussed. Finally, the architectural principles for the data transformation are applied in a practical workflow for the transformation process.

3.1 | Architecture

A straightforward solution for the transformation would be to take a snapshot of the source data store (D_{src}), transform the snapshot, and load it into the transformed data store (D_{trans}). No queries would be allowed during the transformation process, effectively shutting down any data store operations by applications. However, in production environments, it is important that any live application supported by the data store encounters no or minimal impact on their operations. Queries submitted by the application after the snapshot of D_{src} was taken could still be executed on D_{src}, but in order for D_{trans} to contain the latest data and/or reflect the latest changes to its data and structure, queries that insert new or modify existing data need to be transformed as well. The transformation process can therefore be divided in 2 parts: the transformation of a snapshot of D_{src} and the transformation of the data inserted or modified by queries arriving after the snapshot of D_{src} was taken. The specific time when the snapshot of D_{src} is taken is indicated by T_{snap}. Important to note is that all queries will still be executed on D_{src} during the following transformation to support any live applications.

Transforming the D_{src} snapshot into D_{trans} can be regarded as a batch job. It has access to the entire data set, ie, the snapshot of D_{src}, and processes the transformation of this entire data set. Once this batch job is finished, D_{trans} still requires to be updated with the new and adjusted data that is contained within the queries that arrived after T_{snap}. An obvious choice would be to rerun the batch job for these queries. However, during this second transformation, new queries would possibly still arrive as well, requiring yet another run of the batch job. Depending on the arrival rate of the queries, the batch job would run on an ever-reducing set of queries, decreasing the performance of these runs because of a static overhead.[27] In a worst case scenario, it would never reach a consistent state. A better solution would be to use a streaming analysis component, transforming the queries in parallel to the batch transformation as soon as they arrive. The additional benefit of this streaming layer is the continuous query transformation it can provide after the changeover to D_{trans} is complete. Continuous query transformation is a situation where the application would be able to query D_{trans} in the query language of D_{src} through the live transformation in the streaming layer. This effectively eliminates any changes to the application.

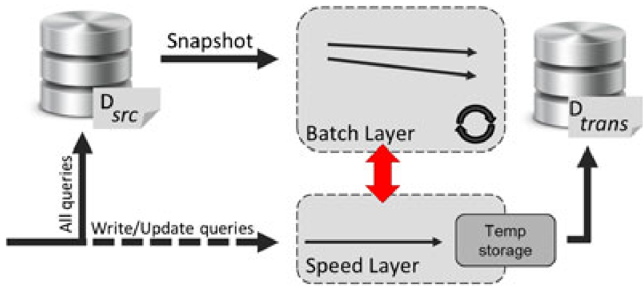

FIGURE 1 General overview of the described architecture with a batch layer and parallel streaming layer

Such a 2-layered hybrid solution is often referred to as the Lambda architecture, a term coined by Nathan Marz.[28] The concept leverages the computing power of batch processing with the responsiveness of a real-time computation system. However, the solution described in this paper defers from this concept in an important way. In the original Lambda architecture, the batch layer continuously reanalyzes an increasing big data set, whereas the proposed solution uses the batch layer for one iteration only, ie, the transformation of the snapshot of D_{src}. The Tengu platform provides a generic implementation of the Lambda architecture, independent from the technologies used for the different layers.[10] The solution described in this paper will therefore be deployed on the Tengu platform.

Figure 1 shows a general overview of the proposed architecture. The batch layer uses a snapshot to transform the structure and data present in D_{src} at T_{snap}, while the streaming layer transforms queries that add new data or transform existing data or structure. The latter transformations are stored in sequence until the batch layer is finished, after which the queries are executed on the newly created D_{trans}. Again, all queries arriving after T_{snap} are still being executed on D_{src} as well, since the latest data needs to be readily available for the application during the transformation. Once the batch layer is finished and while the stored queries from the streaming layer are executing on D_{trans}, a changeover process is started. This changeover process stops all queries from being sent to D_{src} and completes the changeover to D_{trans}. Figure 2 shows the sequence diagram of all the architectural components.

3.2 | Transformation approach

Two main approaches can be identified when looking at the actual transformation of a data store: direct transformation and transformation through a centralized data model. The first is fairly straightforward as one data store is directly mapped onto another. Unique properties of a certain data store can be mapped onto specific traits of the other entirely. However, for each new supported data model, this approach would require a new implementation for transforming the new data model into each of the already supported models. For example, when a transformation is needed between SQL and Cassandra, a

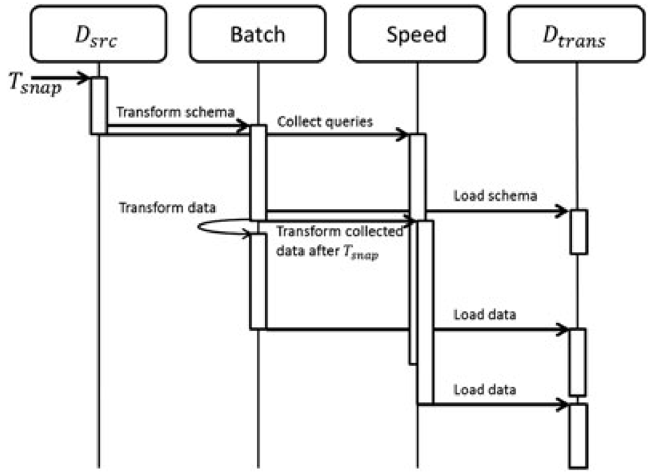

FIGURE 2 Sequence diagram detailing the functionality of the architectural components during the transformation

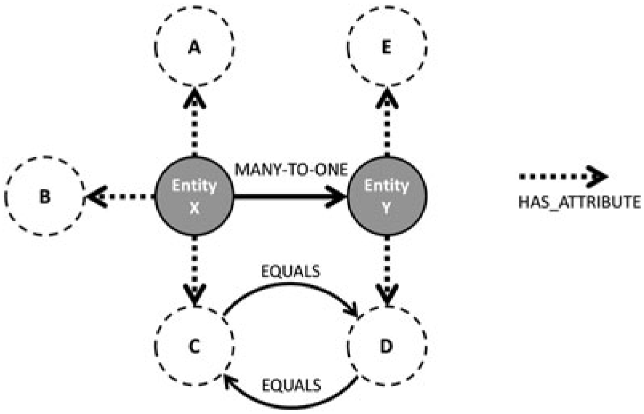

FIGURE 3 Canonical model for the structure of a data set

direct transformation can be implemented in both directions, but when support for MongoDB is required, a transformation also needs to be implemented for both SQL and Cassandra. The amount of effort to support new data store technologies would only grow exponentially. Using a centralized data model would solve this issue by first transforming the structure and data of each data store to the data model, after which it is transformed into the new data store. Supporting new data stores would then only require a transformation towards and from the centralized data model. While this solution does support the extensibility of additional data stores being added, it also has several drawbacks. Firstly, the solution requires an extra transformation for every conversion between data stores introducing additional overhead. Secondly, while transforming to the centralized data model, it is not possible to assume anything about the unique characteristics of D_{trans} as the destination data store is not yet known at that point.

Within the centralized data model, 2 possibilities exist: an abstract or a canonical model. An abstract model can represent the most common characteristics shared by several data stores, while the canonical model aims to support every specific characteristic of each supported data store. Although the abstract data model allows a general representation of the data store's structure and data, not all unique characteristics of the data stores can be supported and any related advantages are also lost. With this in mind, the approach with a canonical model is preferred. The complexity in developing such a solution is mostly contained in the first stage. Once the canonical model is in place, adding support for new data stores is significantly easier. Even if this approach performs worse time wise, compared to a direct transformation, the architecture proposed in Section 3.1 still allows for the application to operate with minimal impact. That is, during the transformation, D_{src} is still the main data store, ie, it still processes all the queries from the application, while the streaming layer transforms any queries that update or insert data in the data store.

The canonical model can be represented through a directional graph, clearly showing the relations between elements of the data model. This graph representation also allows reasoning on the data model as it mimics the properties of an ontology, a model describing a domain in classes with properties and relations.[29] The domain in this case is the data model of D_{src} and the reasoning allows for insights as the data model of D_{trans} is built. Figure 3 represents an example of a canonical model for the structure of a data set. The central element in this canonical model is the Entity. It represents a subject and is built up by different Attributes. An Entity also keeps information about its identifiers and Attributes through a relation *HAS_ATTRIBUTE*. Another type of relation, *EQUALS*, indicates that 2 attributes contain the same information. Relations between Entities can also be represented with a specific type, such as *ONE_TO_ONE*, *MANY_TO_ONE*, and *MANY_TO_MANY*. While the data is not mentioned in Figure 3, it can be regarded as a combination of singular pieces of information, related to attributes as part of an entity (eg, a row in an SQL table or a document in MongoDB).

In a previous publication, the canonical model was represented in an extended entity relationship (EER)–like model.[8,30] This approach was however later found to be too constricting for the canonical model. The current representation of the canonical model as a directional graph allows for extensive reasoning similar to ontologies, which was not possible with the EER-like model. This will aid in the detection of relationships in the data schema and the transformation from the canonical model to D_{trans}. Moreover, while there is no way to prove the soundness and completeness of this representation, it is based on the relational algebra and is much easier to extend if a new type of relationship would be needed.

3.3 | Workflow

This section summarizes the typical workflow of a transformation by the framework. The transformation process can be described in 4 steps:

1. **Initiate transformation:** The transformation is initiated, based on monitoring data or by request. A snapshot is taken from D_{src} and passed on to the batch layer. Until the handover, the final step, D_{src}, acts as the main data store for the application(s), ie, all queries are still passed on to this data store. However, all queries that insert or update data in the data store are also forwarded to the streaming layer as soon as the snapshot is initiated. Currently, queries that alter the schema of D_{src} are not allowed during the transformation process.
2. **Transform schema:** Before the data can be transformed, the batch layer transforms the structure or schema of D_{src}. The streaming layer is only collecting queries, but not yet transforming them, as information is needed about the transformed schema of the data store.
3. **Transform data:** Based on the transformed schema of D_{src}, a new data store, D_{trans}, is set up. Data from the D_{src} are transformed in the batch layer, while recent queries that were collected in the streaming layer are transformed as well. However, resulting transformed queries from the streaming layer are only inserted in D_{trans} after the transformed data from the batch layer has been inserted into D_{trans}.
4. **Handover:** As soon as the data from the snapshot are transformed and put into D_{trans}, the handover is initiated. All queries are then redirected to D_{trans} with respect to any queries still in queue at the streaming layer.

At this point, the application still queries in the language of D_{src}, which leads to the following possible scenarios:

- The application maintains the original language and every query is translated by the streaming layer. The application thus remains dependant on the proposed architecture with a minimal overhead introduced by the continuous transformation.
- The application was prepared for this transformation and changes its querying language to that of D_{trans}.
- The application communicates to the data store through an abstract data layer, such as Hibernate ORM/OGM, PlayORM, or Apache Drill.

It is clear that to eliminate the need for the application to change, the continuous transformation of the queries is required. In practice, this translates to the transformation of data retrieval queries, such as SELECT queries in SQL. Section 4.3 details the transformation of data retrieval queries to eliminate the need for applications' redesigns.

4 | TRANSFORMATION ALGORITHM

4.1 | Schema queries

An overview of the transformations to and from MySQL, Cassandra, and MongoDB is given below. The transformations of MySQL and Cassandra have been detailed in a previous paper,[8] but since the canonical model has changed from an EER-like model to graph representation, the implementation has been completely redone. However, the transformation rules below are still valid and similar to the ones described in Vanhove et al[8]; therefore, only a summary of the transformations for MySQL and Cassandra is given.

4.1.1 | SQL transformations

The SQL[31] is a language for managing relational databases based on relational algebra. Structured Query Language is a standard of both the American National Standards Institute and the International Organization for Standardization.[31] The popularity of SQL has spawned many dialects for its different implementations, such as MySQL, Microsoft SQL Server, and PostgreSQL. The transformation detailed below only uses elements from the SQL standard.

To canonical

The following schema shows how the different data structures from SQL are mapped onto the canonical data model:

The first 2 transformations are straightforward: A table is a collection of columns, similar to an entity with its attributes. In SQL, the relationships are defined through foreign keys, primary keys, and table use. Three types of relationships exist: one-to-one, many-to-one, and many-to-many. The canonical model has an explicit representation of these relationships, therefore, the objective of the transformation is detecting the context of the foreign keys as defined by the SQL standard[31] and translating to the correct relation type.

From canonical

Similar to the transformation towards the canonical model, entities are translated into tables with columns based on the

TABLE 2 Transformation schema from SQL to canonical model

SQL	Canonical
Table	Entity
Column	Attribute
Foreign keys	Relations

Abbreviation: SQL, Structured Query Language.

attributes. These transformations are the exact opposite of those listed in Table 2. The relations are implemented using the foreign keys according to the SQL standard.[31]

4.1.2 | Cassandra transformations

Cassandra is a column-oriented data store originally developed at Facebook.[32] While showing many similarities with classic databases, it does not support a full relational data model. It is aimed at large-scale implementations across hundreds of physical servers with high-availability services.

To canonical

Columns, grouped in column families, are the building blocks of a Cassandra data store, similar to columns and tables in SQL, respectively. This similarity is continued in the translation towards the canonical model, where column families become entities and columns become attributes.

Important to note is that there is no explicit way to infer relations from the Cassandra data model, due to the lack of support for a relational data model. There are however several indicators that relations are present in the Cassandra data model: *index column families*. These column families containduplicate columns from 2 or more column families involved in the relationship and are identified by a primary key that spans multiple columns, also referred to as a *composite key*, containing the primary key data of the related column families. This denormalization eliminates the need for join-like queries (cf SQL) optimizing the data store for read performance. While their presence is a good indicator, composite keys are also used for other purposes, such as column family sorting. This makes it significantly difficult to define a generally automated way of detecting relations in Cassandra. Currently, the automated detection of relations from Cassandra is not supported unless the following naming is used for the index column family "*<entity x>_<entity y>_index*".

From canonical

Translating into Cassandra from the canonical data model, entities are transformed into column families with columns defined by the attributes. As mentioned before, the Cassandra data model does not allow for the explicit representation of relations, but it is possible to represent them through index column families with composite keys. Relations in the canonical model trigger the creation of these additional index column families containing data from the entities involved when translating into Cassandra. The transformations in Table 3 are the exact reverse.

TABLE 3 Transformation schema from Cassandra to canonical model

Cassandra	Canonical
Column family	Entity
Column	Attribute
Index column families	Relations

4.1.3 | MongoDB transformations

An additional NoSQL data store was added to the list of supported data stores: MongoDB is a document-based data store.[33] It stores data as a key paired with a document containing key-value pairs, key-array pairs, or even nested documents. MongoDB accepts JSON documents as data for its collections, represented as binary-encoded JSON behind the scenes.

Listing 1 shows an example of a JSON document with several different elements. First and foremost, a document has fields linked to values. A field can have a single value with familiar data types, such as integers and strings, but also contain an array of values (eg, "network info" in Listing 1) or even embedded documents (eg, "device" in Listing 1). MongoDB is praised for its flexibility as collections do not impose any data model on the stored JSON documents. This lack of data model has some significant effects on the complexity of the transformation.

To canonical

In SQL and Cassandra, a dump of the data store includes the schema or model, ie, every tuple (or row) of data is defined by a certain amount of attributes (or columns). This means the structure of the data is known without even looking at the data itself. MongoDB has a flexible data schema in its collections, ie, collections do not enforce document structure. It is therefore impossible to know all the attributes

Listing 1: Example of a document in JSON

```
{
 "id": 00001,
 "device_id": 00023,
 "device":
  {"device_id": "00023", "platform": "cisco", "name": "router"},
 "network_info":[00001, 00002, 00003, 00004]
}
```

TABLE 4 Transformation schema from MongoDB to canonical model

MongoDB	Canonical
Collection	Entity
Document field	Attribute
Embedded Document	One-to-one relation Many-to-one relation
Document referral array	One-to-many relation
Embedded document array	Many-to-many relation

of the documents without looking at the documents themselves. To derive the canonical model from a MongoDB data store, all data in the collections need to be checked. For each document in each collection, a list of all the keys needs to be made that represent the attributes of the entity in the canonical model. It is clear that checking only one document for each collection does not suffice as documents may also include different keys within the same collection. It is to be expected that iterating over the entire data set in MongoDB as to acquire the canonical model will have negative impact on the transformation time compared to SQL or CQL. The schema in Table 4 details the transformation to the canonical model.

The flexible schema also limits the possibility of accurately defining relations between documents and/or entities. References to other documents can be made through document referral, but this is not explicitly mentioned in the document as is the case with foreign keys in SQL. Therefore, there is no way to automatically detect a relation based on a singular document reference in a field. Another way of defining relations between collections however is through embedded documents. If a field in a document contains an embedded document, this can be indicative of a one-to-one or a many-to-one relationship. Additionally, arrays in documents containing multiple references to other documents or containing embedded documents can indicate one-to-many or many-to-many relations. Note that both many-to-one and one-to-many relations are mentioned here. The flexibility of the data model allows us to represent this relationship in 2 ways: denormalized with redundant data stored for low read query latency or through the array data structure for hierarchical data sets. It also becomes clear that a many-to-many relation actually is 2 one-to-many relations between 2 entities directly. Thanks to the array data structure in MongoDB, no additional entity is needed to represent the relationship as is the case in SQL.

From canonical

As for the transformation towards MongoDB, the flexible data schema simplifies the process. Documents contain keys based on a subset of the attributes defined in the canonical model and are added to their collection based on the entity. If a collection does not yet exist, one is made automatically

TABLE 5 Transformation schema from canonical model to MongoDB

Canonical	MongoDB
Entity	Collection
Attribute	Document field
One-to-one relation	Document integration
One-to-one relation	Document referral
Many-to-one relation	Embedded document
One-to-many relation	Document referral array
Many-to-many relation	Embedded document array

in MongoDB. No schema transformation is therefore needed, as all information is derived from the data. The data are transformed in JSON documents and pushed in MongoDB.

As mentioned before, relations between documents and collections cannot be explicitly expressed in MongoDB. Similar to Cassandra, a denormalization of the canonical model can be used to indicate these relations, or similar to SQL, references can indicate a relation based on an id. Depending on the application requirements, a choice can be made to either normalize or denormalize the MongoDB data store. For example, in a hierarchical data set, it would be wise to normalize the data store and work with references, but, if read performance is a nonfunctional requirement, embedding subdocument information in documents yields less queries. The schema in Table 5 details the transformation from the canonical model.

It is important to note that previous subsections describe the transformation to and from the canonical model for 3 specific technologies, but that the algorithm is inherently technology independent. If a new technology were to be supported, a transformation similar to the ones above should be implemented. Once this is done, transformations to and from each already supported technology are possible.

4.2 | Data insertion queries

The previous section discusses the transformation of the schema of a data store, but the data in the snapshot of D_{src} and new data received after T_{snap} need to be transformed as well based on the created canonical model. To maintain the flexibility and extensibility of the implementation, data insertion queries are transformed to an intermediate state called *tuples*. These tuples contain all the key-value pairs contained within the insertion queries. From these key-value pairs, queries are made up for D_{trans}. Below is an example for an INSERT query from SQL (D_{src}) in Listing 2 transformed into MongoDB (D_{trans}) in Listing 3:

The data that is now injected into MongoDB may not yet complete. The column "device_id" in SQL, and corresponding document field in MongoDB, has been defined as

Listing 2: Example of an INSERT query in SQL

```
INSERT INTO log (l_id, registrationtime, eventcount, device_id)
      VALUES ("2583301", "2010/10/28 20:22:10", "1", 30601);
```

Listing 3: Example of an INSERT query in MongoDB

```
db.log.insert({l_id: "2583301", registrationtime: "2010/10/28
   20:22:10", eventcount: "1" ,device_id: 30601})
```

Listing 4: Example of embedding a document in MongoDB

```
db.log.update({device_id: 30601}, { $set: {device: {id: 30601,
   platform: "Cisco",location: "Brisbane",customer: "Thomas"}}})
```

Listing 5: Example of a SELECT query in SQL

```
SELECT log.l_id, device.location FROM log INNER JOIN device
      ON log.device_id=device.id
```

Listing 6: Example of a SELECT query in MongoDB

```
db.log.find({ },{ l_id: 1, device.location: 1 })
```

a foreign key as part of a many-to-one relationship. On the one hand, if the goal is to create a hierarchical data store, a document reference would be sufficient. On the other hand, if query performance is the goal, a more thorough solution would be to store the "device" information as an embedded document. Listing 4 shows the query that updates the document in the "log" collection with an embedded document.

As the aim of this paper is to decrease query latency for legacy applications and data stores, the implementation of the algorithm uses the embedding of documents instead of document referral.

4.3 | Data retrieval queries

As mentioned in Section 3.3, once the handover to D_{trans} is complete, the application is still querying in the language of D_{src}. There are several solutions to resolve this, but this paper's premise is to eliminate application redesign. This means that continuous transformation needs to be implemented, ie, the continuous translations of queries in the query language of D_{src} into queries of D_{trans}. The data insertion queries have been handled in Section 4.2, and in this section, the data retrieval queries will be detailed. Listing 5 shows a standard SELECT query in SQL.

Similar to all queries, a transformation is made towards a generic representation. From this generic representation, a data retrieval query is made for D_{trans} (eg, MongoDB). The complexity in these selections comes from the joins of different entities, but Section 4.2 detailed that embedded documents were used for the representation of relations. The corresponding MongoDB query is written in Listing 6.

Since the document is embedded, it is clear that less calculations are needed to reach the same results. Section 6 shows the impact of the simplified querying.

5 | IMPLEMENTATION DETAILS

5.1 | Technology choice and motivation

As mentioned in Section 3.1, an implementation of the Lambda architecture is used as part of the Tengu platform.[10] As this implementation is technology independent concerning the different layers, a decision needs to be made as to which technologies are used.

The technology used for the batch layer needs to be able to transform a data store from a legacy application efficiently. The MapReduce model, introduced by Google,[34] is one of the best known programming models for Big Data analysis with Hadoop as the implemented open source framework. In the previous implementation of the transformation, algorithm MapReduce on Hadoop was used.[8] However, Spark is considered to be the successor of Hadoop MapReduce with execution times 10 up to 100 times faster through in-memory computing.[35]

For the streaming layer, many of the previously mentioned technologies for the batch layer have (near) real-time streaming variants. Although many support streaming, for as many this has never been the sole focus. Storm, on the other hand, is an analysis framework entirely built around the idea of (near) real-time analysis of streams. It was originally developed at Twitter and is now part of the Apache project. This aside, implementing Storm as part of this proof-of-concept, clearly shows that both layers can be entirely different and

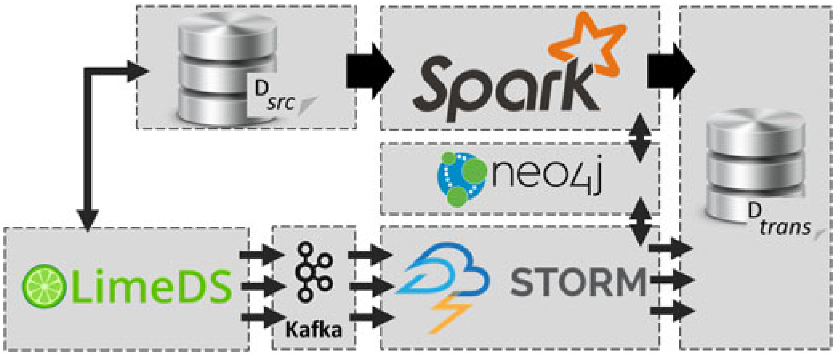

FIGURE 4 Instantiation of the framework with all the implemented technologies

independent technologies. Both Spark and Storm use Java, which means code is reusable across both layers.

An overview of all the integrated technologies is shown in Figure 4. Aside from the aforementioned analysis technologies, several supporting technologies are mentioned as well. The LimeDS framework allows for wiring different data-driven services together in an easy way.[36] In this implementation, it is responsible for directing queries to D_{src} and the streaming layer. It also manages the synchronization between the batch and streaming layers. A second supporting technology integrated in the implementation is Apache Kafka.[37] Kafka is a publish-subscribe messaging system implemented as a distributed commit log. Storm and Kafka naturally work well together, as both have strong guarantees on message processing. Finally, the canonical model is stored in a graph data store. In this particular implementation, Neo4j was chosen for its maturity, extensive documentation, solid performance, and supporting community.[38]

5.2 | Transformation algorithm

The pseudocode in Algorithm 1 can be clearly divided in 2 parts: the schema transformation and the data transformation. During the schema transformation, schema queries from the D_{src} snapshot are translated into the canonical model and stored in Neo4j. From this representation, the schema is built for D_{trans}. Once the D_{trans} schema is ready, the transformation of the data in the D_{src} snapshot is started. All data queries are first translated to a generic tuple representation based on the canonical model and then matched on the D_{trans} schema. The additional step of translating to a generic representation is necessary for the data as well as to maintain code independence between data store technologies.

The code also contains several for loops, but these loops are distributed and executed across the entire Spark cluster in parallel. This is especially important for the transformation of the data contained within the snapshot, as the schema information in a snapshot is negligibly small in most cases compared to the data. Each slave in the cluster gets a small subset of the D_{src} snapshot and transforms this subset towards the canonical model and on from the canonical model to D_{trans}.

Algorithm 1 Pseudocode describing the schema and data transformation in the batch layer

```
read snapshot D_src
for each schema query in the snapshot do
    transform to canonical model
end for
store schema in canonical model
detect relations in canonical model
for each entity in canonical model do
    transform to D_trans
end for
load schema in D_trans
for each data query in the snapshot do
    transform to tuple in canonical model representation
end for
for each tuple do
    transform to D_trans
end for
load data in D_trans
```

Parsing the query language of D_{src} in the transformation to the canonical model is done through ANTLR.[39] ANTLR generates a parser/lexer in Java, based on a grammar file containing a description of the structure of the language to be parsed. In this paper, grammar files were used for MySQL and the CQL. There is no grammar for MongoDB, as it uses JSON to store the documents in its collections and no data model is forced upon the documents.

6 | EVALUATION

6.1 | Experimental setup

The implemented instantiation of the architecture was deployed on the Virtual Wall. The iLab.t Virtual Wall facility[†] is a generic test environment for advanced network, distributed software, and service evaluation and supports

[†]http://ilabt.iminds.be/

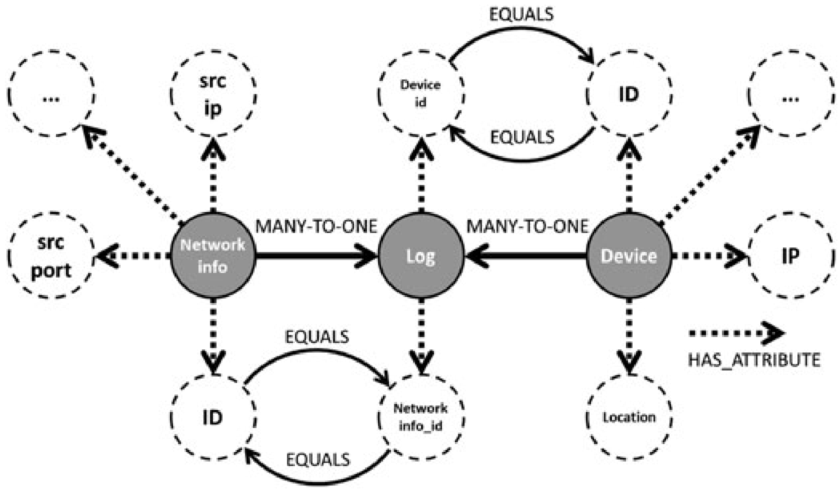

FIGURE 5 Partial canonical model of the network logging data store

scalability research. The Virtual Wall contains 300 nodes with varying hardware specifications. The server specifications in these experiments were as follows: Dual CPU (Quad core) with 12 GB of RAM and 1 × 225 GB disk. Four physical nodes were used for a dedicated 3-worker Spark cluster, 2 nodes for the Apache Storm cluster, and single-node instances for MySQL, MongoDB, Neo4j, and a Cassandra data store.

6.2 | Use case description

This use case shows the application of the transformation algorithm on a data store containing network logging information. Currently, the network monitoring platform uses a relational data store in MySQL to save information, but the query performance is no longer sufficient, for real-time querying and feedback as responses take several minutes. The aim is to lower query latency by transforming the MySQL relational datastore (D_{src}) into one of the supported data store technologies, whichever yields better results. As a reference, Figure 5 shows a partial canonical model after the transformation from MySQL. Three entities can be identified: device, network_info, and log. Device contains information about a certain network device, such as a router, while network info stores information about the logged package, containing amongst others a source/destination ip and port and a protocol. An example of the log can be seen in Listing 1. The sizes of the data sets used for the evaluation contain 100 000, 5 million, 10 million, and 15 million logs.

6.3 | Results

Spark resource tuning

Spark has a large number of configuration parameters that influence resource utilization with drastic results on execution performance. Figure 6 shows the execution times for the transformation of the SQL snapshot described in Section 6.2 with varying data set sizes expressed as a number of logs and for different configuration parameters. The impact of the configuration parameters can be clearly seen in the graph. For example, a snapshot containing 5 million logs is transformed in around 2 hours (129 minutes) with 2 executors, each having access to 6 GB of memory and 8 cores, while the same snapshot only requires around 27 minutes of execution time with 20 executors with 1 GB and 8 cores. Both executions do however use the maximum memory resources available in the entire cluster, taking into account the standard limitations defined by the Spark cluster, but the distribution of the resources also factors in. Spark thrives on in-memory computing, but for the computation of the transformation algorithm, it clearly does not require 6 GB of memory per executor. One gigabyte is enough for 20 parallel executors to outperform the previous configuration given the size of the snapshot. Allocating too much memory to an executor often results in excessive garbage collection delays, which can be clearly

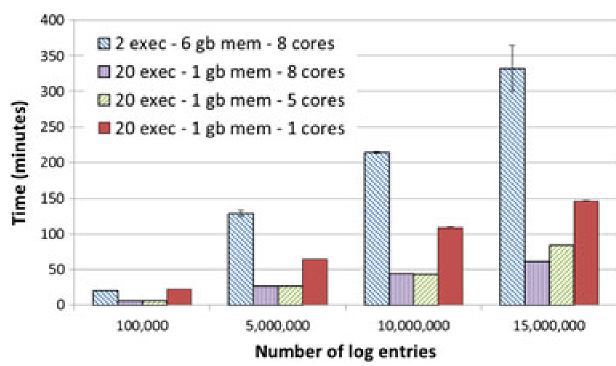

FIGURE 6 Graph showing the transformation time of an SQL snapshot data store to MongoDB for different Spark configuration parameters. They express the amount of parallel executors that are used and how much memory and how many cores are available to each executor. SQL, Structured Query Language

seen in these results. Moreover, when working with a larger number of executors, the standard deviation remains smaller because delays in a specific executor can be easily caught by the other remaining executors.

Additionally, the amount of cores also influences the parallelism in Spark. When using Hadoop Distributed FileSystem (HDFS) with Spark, it is recommended to not use more than 5 cores per executor, as HDFS does not deal well with excessive amounts of concurrent threads.[40] Limiting the amount of cores per executor to 5 keeps the execution time at an average of around 27 minutes. So even though the parallelism is decreased, the execution time remains the same. However, when a larger number of logs is considered, eg, 15 million, the 8 cores execution outperforms the 5 cores configuration. For this data set size, it seems HDFS is still able to scale, but with growing data sets, it is important to take account of this parameter.

Decreasing the number of cores even further to only 1 core per executor increases the transformation time back to around 1 hour (64 minutes) for 5 million logs as only a limited amount of tasks are allowed to execute in parallel. Figure 6 clearly shows the impact of Spark resource tuning on the transformation time. The influence of these parameters on memory management in Spark has also been extensively researched in previous papers.[40,41] In general, increasing memory size in a Spark cluster will lower the execution time linearly until the entire data set is able to be loaded in memory. Increasing the memory size further will introduce garbage collection delays as seen in Figure 6. For the algorithm in this paper with the described use case, a solid configuration was found for 20 executors each with 1 GB of memory and 8 cores. However, when scaling to larger data stores, these parameters need to be optimized continuously to achieve the best performance.

Snapshot transformation performance

Figure 7 shows the execution times for the transformation of the SQL snapshot described in Section 6.2 to MongoDB, Cassandra (CQL), and SQL with varying data set sizes expressed as a number of logs. The transformation towards SQL is used to check the correctness of the algorithm as we expect to get an exact copy of the snapshot. All execution times show a linear trend with the increasing data set size, but the execution time of a transformation towards to MongoDB is significantly smaller compared with SQL and CQL. As described in Section 4.1.3, MongoDB is praised for its flexibility because collections do not impose any data model on the stored documents. While in a generic transformation, information from the canonical model is used to construct a data model for D_{trans} and to make sure data adheres to this data model, this is less so for MongoDB as no strict data model is required. The canonical model is stored in Neo4j, so to retrieve this information, a connection to this data store is needed. MongoDB limits the number of connections that are required during its transformation from the canonical model, therefore lowering the total execution time. Moreover, the amount of queries generated while transforming to CQL is higher compared with MongoDB and SQL because of the denormalization of data stored in the index column families representing the many-to-one relations as mentioned in Section 4.1.

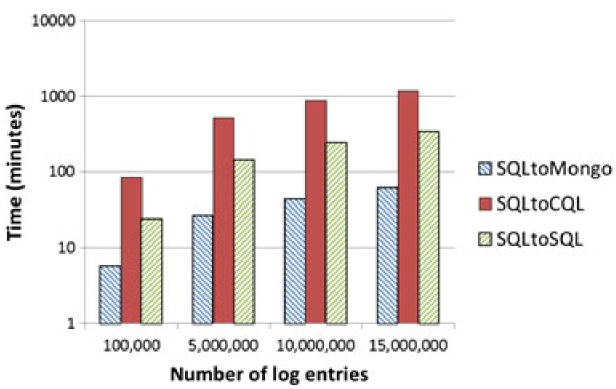

FIGURE 7 Graph showing the transformation time of an SQL snapshot data store to MongoDB, Cassandra (CQL), and SQL with 20 Spark executors each with 1 GB of memory and 8 cores. CQL, Cassandra Querying Language; SQL, Structured Query Language

Query latency performance

The ultimate goal of the transformation is to reduce query latency by transforming schema and data to a different data store technology. Figure 8 shows the average query latency for a JOIN

query in SQL requesting all the logs joined with the information from the devices. This is a very expensive operation in SQL causing an exponential growth of the execution time with growing data sets. The JOIN query in SQL can however be translated into a selection query in MongoDB, still returning the same data, as all data from the many-to-one relation is embedded in the documents. The denormalization that was

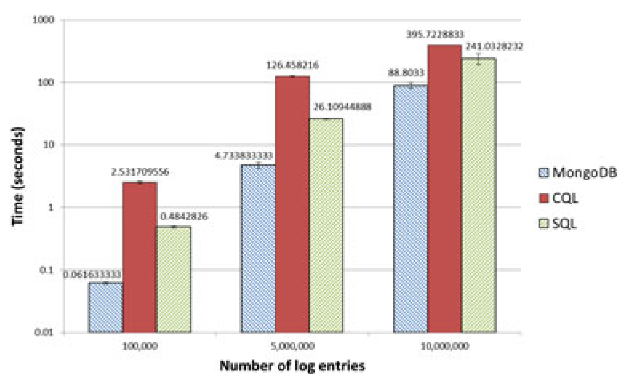

FIGURE 8 Query latency for JOIN-like query in different data stores: SQL, Cassandra (CQL), and MongoDB. CQL, Cassandra Querying Language; SQL, Structured Query Language

TABLE 6 Average execution time of different Storm bolts for the transformation of SQL queries, through the SQLMapBolt, into MongoDB, Cassandra (CQL), and SQL

	SQLMapBolt	MongoReduceBolt	CQLReduceBolt	SQLReduceBolt
Execution time	37.568 ms	14.458 ms	80.133 ms	79.112 ms

Abbreviations: CQL, Cassandra Querying Language; SQl, Structured Query Language.

introduced by the transformation pays off in query latency as only one entity of a data store technology needs to be consulted to retrieve the same data. Similarly, in Cassandra, the same data can be retrieved by querying the index column family that represents the many-to-one relationship. However, not using a partition key to retrieve data from a column family is a heavy operation in Cassandra, eliminating it from consideration for this use case.

Continuous transformation performance

After the transformation of the snapshot and the handover, the application still queries in the language of D_{src}. Considering the use case, while there is a significant time gain transforming the data store to MongoDB or Cassandra, the overhead of transforming queries from the application needs to be limited to benefit from this transformation. The evaluation of the continuous transformation was performed on a 2-node Apache Storm cluster with standard configuration where each bolt was assigned a single worker in the cluster. Table 6 shows the average transformation time of a single query in every step of the transformation. For example, in a transformation from SQL to MongoDB, a query would pass through the SQLMapBolt, mapping the query onto the canonical model, after which it will be reduced towards MongoDB by the MongoReduceBolt. This yields a total overhead of 52.026 ms (37 and 14 ms, respectively). Considering the query performance from Figure 8, it is clear that the results of the transformation approach in this paper benefits the application's query latency, increasing the general performance.

6.4 | Discussion

The results in Section 6.3 clearly show the ability of the proposed algorithm to transform schema and data of a data store into a technology that yields better query latency performance as well as support for continuous transformation of application queries within a reasonable time frame. While several limitations to the current system exist, this section discusses those limitations and provides possible solutions on how to mitigate them.

The approach of the proposed algorithm in this paper, detailed in Section 3.2, is theoretically slower than the direct approach, as it requires one additional transformation to or from the canonical model. Direct transformations are however less extensible towards future technologies as support for a new data store technology requires an entirely new implementation to transform to and from each existing technology. The Schema Conversion Tool provided by Amazon, discussed in Section 2, can be regarded as a bundle of direct transformations between dialects of SQL. This tool may achieve a faster performance compared with the approach described in this paper, but contrary to the proposed approached, only SQL dialects are currently supported and extending the tool would require an entirely new code base. Moreover, the additional latency introduced by the described approach in this paper is alleviated by the use of the Spark platform. Spark allows for in-memory computing yielding faster execution times but its clustered architecture also allows for scaling towards specific time constraints with minimal effort.[41]

A second limitation is that relations between entities in the canonical model are determined by the explicit and implicit use of certain data structures in the data schema (eg, foreign keys in SQL, composite keys in Cassandra, and arrays in MongoDB). However, it is conceivable that the implicit use of these data structures may not always be found, especially in NoSQL data stores such as Cassandra and MongoDB. Given that specific situation, the algorithm would currently only detect the entities for its canonical model with no relations between them. While still being able to transform these entities to another data store technology, it might not yield a better query latency performance. An interesting extension of the algorithm would therefore be an automated detection of relations in the canonical model based on the read queries effectively optimizing the data schema based on its use. For example, SQL retrieval queries with JOIN operations indicate a relationship even if foreign keys were not defined. For NoSQL stores, this needs to be derived from the sequence of queries that are often requested in succession. These chains of queries indicate the potential existence of a relationship between the entities. The extension of the algorithm to automatically detect relationships in the canonical model based on querying behavior is deferred to future work.

Finally, the current algorithm is not equipped to deal with queries that alter the data schema of D_{src} while the transformation process is in progress. While creating the data schema of D_{trans}, any changes to the schema of D_{src} would potentially create inconsistencies while adding the data to D_{trans}. It was therefore decided to deny any queries that alter the schema until the handover is completed. The continuous transformation could then deal with the schema altering queries, which will reflect in both D_{trans} as in the canonical model.

7 | CONCLUSION AND FUTURE WORK

This paper introduces an approach and algorithm for schema and data transformation as a means to support dynamic data storage and polyglot persistence. The approach uses an intermediate canonical model to ensure the flexibility and extensibility of the implementation towards future supported technologies. To support a new data store technology, one only needs to implement a transformation towards and from the canonical model. In previous work, support for SQL and CQL was already discussed, but the implementation has been revised as part of the newly changed canonical model. The paper also introduces support for MongoDB, a NoSQL document data store. The transformation algorithm is implemented as a Lambda architecture with a batch and speed layer to support live applications without downtime and the need for code changes. A network monitoring platform is considered as a use case and shows a significant performance increase after the transformations to both CQL and MongoDB. The overhead introduced for the continuous transformation is limited to a maximum of around 100 ms. The time to transform a snapshot heavily depends on D_{src} and the chosen D_{trans} and is influenced by the strictness of the data models.

For future work, now, a transformation algorithm has been defined and implemented, an interesting application would be to fully support dynamic data storage regarding supported implementations, ie, an automated system that stores data in the most optimal format at any given time. Additionally, while data relations are now inferred from defined uses of structures in a data store technology (eg, foreign keys, composite keys, and arrays), the best way to learn the relations in a data set is through its use. Future work will also focus on detecting relations in the canonical model based on reading queries. These changes will be reflected in the transformed data store with the ultimate goal of increasing query performance even further.

ACKNOWLEDGEMENTS

The work in this paper has partly been funded by the iMinds SEQUOIA research project.

REFERENCES

1. Cattell R. Scalable SQL and NoSQL data stores. *SIGMOD Rec*. 2011;39(4):12–27. https://doi.org/10.1145/1978915.1978919.
2. Li Y, Manoharan S. A performance comparison of SQL and NoSQL databases. *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, Victoria, B.C., Canada: IEEE; 2013:15–19.
3. Nance C, Losser T, Iype R, Harmon G. NoSQL vs RDBMS: Why there is room for both. *SAIS 2013 Proceedings*, Savannah, Georgia, USA; 2013.
4. Shute J, Vingralek R, Samwel B, et al. F1: A distributed sql database that scales. *Proc VLDB Endow*. 2013;6(11): 1068–1079. https://doi.org/10.14778/2536222.2536232
5. Corbett JC, Dean J, Epstein M, et al. Spanner: Googleś globally distributed database. *ACM Trans Comput Syst*. 2013;31(3):8:1–8:22. https://doi.org/10.1145/2491245
6. Sadalage PJ, Fowler M. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. 1st ed. Boston, MA, USA: Addison-Wesley Professional; 2012.
7. Sellami R, Defude B. Using multiple data stores in the cloud: Challenges and solutions. In: Hameurlain A, Rahayu W, Taniar D, eds. *Data Management in Cloud, Grid and P2P Systems*, Lecture Notes in Computer Science, vol. 8059: Springer Berlin Heidelberg; 2013:87–98. https://doi.org/10.1007/978-3-642-40053-7_8.
8. Vanhove T, Van Seghbroeck G, Wauters T, De Turck F. Live datastore transformation for optimizing big data applications in cloud environments. *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, Ottawa, Canada: IEEE; 2015:1–8.
9. Vanhove T, Van Seghbroeck G, Wauters T, De Turck F, Vermeulen B, Demeester P. Tengu: An experimentation platform for big data applications. *ICDCS Workshops*. IEEE; 2015:42–47. http://dblp.uni-trier.de/db/conf/icdcsw/icdcsw2015.html#VanhoveSWTVD15. Accessed April 18, 2017
10. Vanhove T, Van Seghbroeck G, Wauters T, Volckaert B, De Turck F. Managing the synchronization in the Lambda architecture for optimized big data analysis. *IEICE Trans*. 2016;99-B(2): 297–306.
11. Shu NC, Housel BC, Taylor RW, Ghosh SP, Lum VY. Express: a data extraction, processing, and restructuring system. *ACM Trans Database Syst (TODS)*. 1977;2(2):134–174.
12. Vassiliadis P. A survey of extract–transform–load technology. *Int J Data Warehous (IJDWM)*. 2009;5(3):1–27.
13. Settlemyer BW, Dobson JD, Hodson SW, Kuehn JA, Poole SW, Ruwart TM. A technique for moving large data sets over high-performance long distance networks. *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, MSST '11. IEEE Computer Society, Washington, DC, USA; 2011:1–6.
14. Zheng J, Ng TSE, Sripanidkulchai K. Workload-aware live storage migration for clouds. *SIGPLAN Not*. 2011;46(7):133–144. https://doi.org/10.1145/2007477.1952700
15. Zhang L, Wu C, Li Z, Guo C, Chen M, Lau FCM. Moving big data to the cloud: an online cost-minimizing approach. *IEEE J Sel Areas Commun*. 2013;31(12):2710–2721.
16. Das S, Nishimura S, Agrawal D, El Abbadi A. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc VLDB Endow*. May 2011;4(8):494–505. http://dl.acm.org/citation.cfm?id=2002974.2002977.
17. Elmore AJ, Das S, Agrawal D, El Abbadi A. Zephyr: live migration in shared nothing databases for elastic cloud platforms. *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD'11. ACM, New York, NY, USA; 2011:301–312. https://doi.org/10.1145/1989323.1989356
18. Ting K, Cecho JJ. *Apache Sqoop Cookbook*. Sebastopol, CA, USA: O'Reilly Media, Inc.; 2013.
19. Rahm E, Bernstein PA. A survey of approaches to automatic schema matching. *VLDB J*. 2001;10(4):334–350.
20. Bellahsene Z, Bonifati A, Rahm E, et al. *Schema Matching and Mapping*, vol. 57. New York City, NY, USA: Springer; 2011.
21. Schildgen J, Lottermann T, Dessloch S. Cross-system NoSQL data transformations with NotaQL. *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for Mapreduce and Beyond*, BeyondMR'16. ACM, New York, NY, USA; 2016:5:1–5:10. https://doi.org/10.1145/2926534.2926535.
22. Liao YT, Zhou J, Lu CH, et al. Data adapter for querying and transformation between SQL and NoSQL database. *Future*

Gener Comput Syst. 2016;65:111–121. https://doi.org/10.1016/j.future.2016.02.002. http://www.sciencedirect.com/science/article/pii/S0167739X16300085, Special Issue on Big Data in the Cloud. Accessed April 18, 2017

23. Mongify. http://mongify.com/ Accessed March 15, 2017.
24. Apache Sqoop. http://sqoop.apache.org/ Accessed March 15, 2016.
25. Leonard A. Hibernate OGM at work. *Pro Hibernate and MongoDB*. New York City, NY, USA: Springer; 2013:51–120.
26. Hausenblas M, Nadeau J. Apache drill: interactive ad-hoc analysis at scale. *Big Data*. 2013;1(2):100–104.
27. Marz N. The mathematics behind Hadoop-based systems. http://nathanmarz.com/blog/the-mathematics-behind-hadoop-based-systems.html Accessed March 15, 2017; 2009.
28. Marz N, Warren J. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Greenwich, CT, USA: Manning Publications Co.; 2014. (Early Access Program).
29. Gruber TR. A translation approach to portable ontology specifications. *Knowl Acquis*. 1993;5(2):199–220.
30. Chen PPS. The entity-relationship model—toward a unified view of data. *ACM Trans Database Syst*. March 1976;1(1):9–36. https://doi.org/10.1145/320434.320440.
31. ISO/IEC 9075-1:2011 Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework). Technical Report, ISO/IEC; 2011. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681. Accessed April 18, 2017
32. Lakshman A, Malik P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Syst Rev*. 2010;44(2):35–40.
33. Chodorow K. *Mongodb: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc.; 2013.
34. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. *Commun ACM*. 2008;51(1):107–113. https://doi.org/10.1145/1327452.1327492
35. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10. USENIX Association, Berkeley, CA, USA; 2010:10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113. Accessed April 18, 2017
36. LimeDS. http://limeds.intec.ugent.be/ Accessed March 15, 2017.
37. Kreps J, Narkhede N, Rao J. Kafka: a distributed messaging system for log processing, NetDB; 2011.
38. Webber J. A programmatic introduction to neo4j. *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, Tucson, AZ, USA: ACM; 2012:217–218.
39. Parr TJ, Quong RW. ANTLR: a Predicated-LL(k) Parser Generator. *Softw Pract Exp*. 1995;25(7):789–810.
40. Ryza S. How-to: Tune your apache spark jobs (part 2). http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/ Accessed March 15, 2017; 2015.
41. Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, San Jose, CA, USA: USENIX Association; 2012:2–2.

Thomas Vanhove obtained his master's degree in Computer Science from Ghent University, Belgium, in July 2012. In August 2012, he started his PhD at the Department of Information Technology (INTEC) at Ghent University, researching data management solutions in cloud environments. More specifically, he has been looking into dynamic big data stores and polyglot persistence. It was during that time he created the Tengu platform for the simplified setup of big data analysis and storage technologies on experimental test beds.

Merlijn Sebrechts graduated in July 2015 as an Industrial Engineer, Informatics, from Ghent University. In August 2015, he joined the Information Technology (INTEC) Department of Ghent University to pursue his PhD. In his PhD, he focusses on cloud modelling languages to solve big data challenges, while remaining an active member of several large open source communities.

Gregory Van Seghbroeck graduated at Ghent University in 2005. After a brief stop as an IT consultant, he joined the Department of Information Technology (INTEC) at Ghent University. On the 1st of January, 2007, he received a PhD grant from IWT, Institute for the Support of Innovation through Science and Technology, to work on theoretical aspects of advanced validation mechanism for distributed interaction protocols and service choreographies. In 2011, he received his PhD in Computer Science Engineering and continued to work at Ghent University as a postdoctoral fellow.

Tim Wauters received his MSc degree in electro-technical engineering in June 2001 from Ghent University, Belgium. In January 2007, he obtained the PhD degree in electro-technical engineering at the same university. Since September 2001, he has been working in the Department of Information Technology (INTEC) at Ghent University and is now active as a postdoctoral fellow of the F.W.O.-V. His main research interests focus on network and service architectures and management solutions for scalable multimedia delivery services. His work has been published in about 70 scientific publications in international journals and in the proceedings of international conferences.

Bruno Volckaert is a professor in the Department of Information Technology (INTEC) at Ghent University. He obtained his Master of Computer Science degree in 2001 from Ghent University, after which he started work on his PhD on data intensive scheduling and service management for Grid computing. His current research deals with reliable and high-performance distributed software systems and clouds.

Filip De Turck leads the network and service management research group at the Department of Information Technology of the Ghent University, Belgium and imec (Interdisciplinary Research Institute in Flanders). He (co)authored over 450 peer-reviewed papers, and his research interests include telecommunication network and service management, efficient big data processing, and design of large-scale virtualized network systems. In this research area, he is involved in several research projects with industry and academia, serves as vice-chair of the IEEE Technical Committee on Network Operations and Management (CNOM), chair of the Future Internet Cluster of the European Commission, and is on the TPC of many network and service management conferences and workshops and serves in the editorial board of several network and service management journals.

How to cite this article: Vanhove T, Sebrechts M, Van Seghbroeck G, Wauters T, Volckaert B, De Turck F. Data transformation as a means towards dynamic data storage and polyglot persistence. *Int J Network Mgmt*. 2017;27:e1976. https://doi.org/10.1002/nem.1976