Live Datastore Transformation for optimizing Big Data applications in Cloud Environments

Thomas Vanhove, Gregory Van Seghbroeck, Tim Wauters, Filip De Turck
Department of Information Technology, Ghent University - iMinds
Gaston Crommenlaan 8/201, 9050 Gent, Belgium
Email: thomas.vanhove@intec.ugent.be

Abstract—**Vendor lock-in is one of the major issues preventing companies from moving their big data applications to the cloud or changing between cloud providers. A choice in provider based on used datastores can be advantageous at first, but with ever-changing applications the chosen datastore may no longer be optimal after some time. Namely, applications' requirements change due to frequent updates and feature requests, and scalability issues arise as user numbers continuously evolve. In this paper we propose a framework for the live transformation of the schema and data of datastores. Using a canonical data model the framework can be easily extended for additional datastores. The framework performs the transformation on two different levels. It uses a batch layer to transform a snapshot of the datastore, while a speed layer transforms queries inserting new or updated data into the datastore. A transformation is given between MySQL and Cassandra as a proof-of-concept. We show the correctness of the transformation and provide performance results, in terms of transformation times and overhead.**

I. Introduction

Vendor lock-in and interoperability issues are still considered to be top inhibitors to cloud adoption, according to a survey by North Bridge among 855 respondents [1]. The choice between cloud providers is in most cases difficult as there are several large players such as Google, Microsoft, and Amazon, but even more smaller players. Comparing these public cloud providers is a tedious task and in order to help future customers decide which cloud provider is best suited for them, tools have been created for the automated comparison of providers based on different requirements [2], [3]. For example, Ruiz-Alvarez and Humphrey have an automated approach of selecting the best storage service for a given dataset of a particular application [3]. Once such a choice is made, the migration to the cloud is a complex process. Firstly, because of the size of current data sets, traditional processing and storage solutions no longer suffice. Working with these big data sets requires parellel software running in clusters of tens, hundreds or even thousands of servers [4]. Secondly, this process usually involves changes to the application, extensive (re)configuration, and/or downtime. But as applications tend to evolve with frequent updates and feature requests on the one hand, and increasing user numbers on the other, their requirements change and scalability issues arise. This leads to a situation where the original, optimal choice of datastore is no longer optimal, i.e. the performance of the applications suffers from this choice. This might call for another migration, even potentially to another provider, again a costly operation.

In this paper, we propose a new framework for live datastore transformation as part of a new Platform-as-a-Service Tengu, previously known as Kameleo [5]. The proposed framework aims to migrate and transform the schema and data of any datastore without any necessary changes to or downtime of the application. It introduces the concept of dynamic storage which allows the stored data to be stored in the optimal format for the application, transforming the format when necessary, i.e. when certain requirements are no longer met (e.g. query time exceeds a certain threshold). This paper shows the extensible approach for transforming datastores live and the architecture to support it. A proof-of-concept implementation is detailed showing the transformation between MySQL, a relational database, and Cassandra, a NoSQL column-oriented datastore. The authors want to emphasize that although datastore is a term often used within the NoSQL domain, while RDBMS prefers the term database, this paper uses datastore as a general term for both.

The remainder of this paper is structured as follows: the architecture of the framework is described in Section II, while Section III details the transformation principles and the corresponding workflow. The algorithm for the transformation is stipulated in Section IV. In Section V, the implementation details of the framework are provided. Section VI details the experimental setup, whereas results can be found in Section VII. The discussion of the results in regard to future work can be found in Section VII-C. Section VIII gives an overview of related work in the field of migration and transformation of datastores. Finally, the main conclusions are presented in Section IX.

II. Architecture overview

When applying a transformation on a datastore, it is important that the live application it supports, encounters no or minimal impact on it's operations. Secondly, the faster a transformation can be completed, the better, as the data is highly susceptible to redundancy and loss in this state. It can be reasonably assumed queries will continue to arrive while the transformation is in progress, considering a live application. Reading information from the datastore during the transformation is straightforward as these queries can be handled by the original or source datastore (D_{src}), but queries inserting new or modifying existing data also need to be transformed, otherwise the transformed datastore (D_{trans}) will

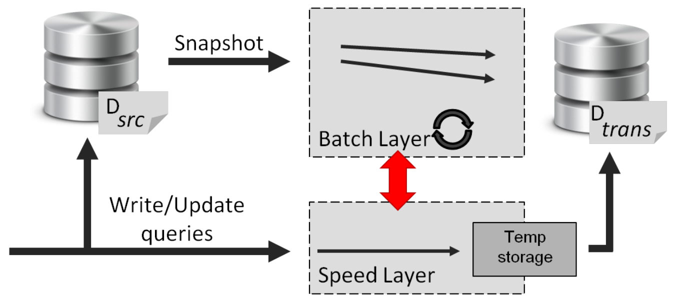

Fig. 1. General overview of the architecture with a batch layer and parallel speed layer.

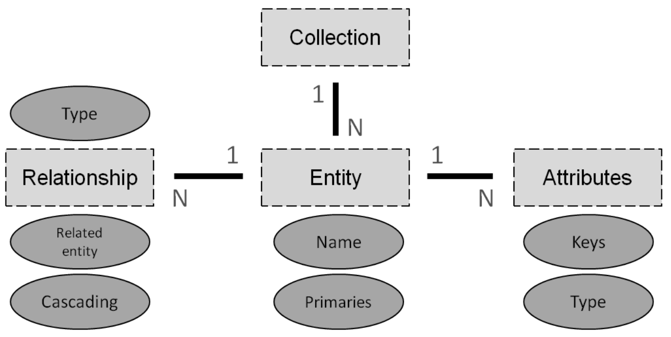

Fig. 2. Canonical model for the structure of a dataset.

not contain the latest data and/or reflect the latest changes to its data and structure. A simple solution would be to store these queries and transform them as soon as the first transformation is finished. However, during this second transformation new queries would possibly still arrive as well, yielding an almost infinite loop. Introducing a real time transformation for these queries, parallel to the batch transformation, solves this issue.

For the sake of completeness, we mention that the Tengu platform already provides a batch and speed layer architecture as a service, the Lambda architecture [5], [6]. It is a specific approach for Big Data analysis leveraging the computing power of batch processing with the responsiveness of a real-time computation system.

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 that time, while the speed 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}. It is important to note that all queries arriving after the snapshot are still being executed on D_{src} as well, since it is still being used for reads. Once the batch layer is finished and while the stored queries from the speed layer are executing on D_{trans}, a changeover process will be started. This stops all queries from being sent to D_{src} and completes the changeover to D_{trans}.

III. Transformation and workflow

A. Approach

Two main approaches can be identified when looking at the actual transformation of a datastore: direct transformation and transformation through a centralized data model. The first approach is fairly straightforward as one datastore is directly mapped onto another. Unique properties of a certain datastore 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. Using a centralized data model would solve this issue by first transforming the structure and data of each datastore to the data model, after which it is transformed into the new datastore. Supporting new datastores would then only require a transformation towards and from the abstract or canonical model. While this solution does support the extensibility of additional datastores being added, it also has several drawbacks. Firstly, the solution requires an extra transformation for every conversion between datastores 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 datastore is not yet known at that point.

Within the centralized data model, two possibilities exist: an abstract and a canonical model. An abstract model can represent the most common characteristics shared by several datastores, while the canonical model aims to support every specific characteristic of each supported datastore. Although the abstract data model allows a general representation of the datastore's structure and data, not all unique characteristics of the datastores are 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 will be mostly contained in the first stage. Once the canonical model is in place, adding support for new datastores is significantly easier. Even if this approach performs worse time-wise, compared to a direct transformation, the architecture proposed in Section II still allows for the application to operate with minimal impact. That is, during the transformation, D_{src} is still the main datastore, i.e. it still processes all the queries from the application, while the speed layer transforms any queries that update or insert data in the datastore.

Figure 2 represents a diagram of a proposed canonical model, based on the Entity-Relationship (ER) model [7], for the structure of a dataset. The central element in this canonical model is the Entity. It represents a subject and is built up by different Attributes. It can be compared to a table in SQL, but, as demonstrated in Section V, not every table in SQL can be mapped onto an Entity. An Entity also keeps information about its primary keys and Attributes. Relations between Entities can also be represented with a specific type, such as many-to-one, many-to-many and one-to-one. Additional information about relationships, such as cascading, can be stored here too. Finally, a Collection combines several Entities, much like the keyspace combines column families in Cassandra. While the tuples (i.e. the actual representation of the data in the datastore) are not mentioned in Figure 2, a tuple can be regarded as a combination of singular pieces of information, related to attributes as part of an entity (e.g. a row in a SQL table).

B. Workflow

This section summarizes the typical workflow of a transformation by the framework. The transformation process can be described in four 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 datastore for the application(s), i.e. all queries are still passed on to this datastore. However, all queries that insert or update data in the datastore are also forwarded to the speed layer as soon as the snapshot is initiated.
2) **Transform structure:** before the data can be transformed, the batch layer transforms the structure or schema of D_{src}. The speed layer is only collecting queries, but not yet transforming them, as information is needed about the transformed schema of the datastore.
3) **Transform data:** based on the transformed shema of D_{src}, a new datastore, D_{trans}, is set up. Both batch and speed layer start transforming the data from the snapshot and queries respectively.
4) **Handover:** as soon as the data from the snapshot is 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 speed 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 speed 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 datastore through an abstract data layer, such as Hibernate ORM/OGM or PlayORM

It is clear that in order to eliminate the need for the application to change, the first option, continuous transformation of the queries, is required. Although we mention the different possibilities here, the further evaluation of these scenarios is outside the scope of this paper and part of future work.

IV. TRANSFORMATION ALGORITHM

As discussed in Section III, based on the canonical model approach, the transformation is divided into two parts: the first part transforms D_{src} into a canonical model and in a second phase from the canonical model into D_{trans}. To clarify the entire process, the transformation is drawn up for two specific datastores. In the context of companies migrating to the cloud and issues like vendor lock-in, an interesting use case is that of a company with a classic RDBMS wanting to migrate to a NoSQL datastore. MySQL, one of the most popular open-source RDBMS solutions, and Cassandra, a popular NoSQL column store, are selected for the proof-of-concept as D_{src} and D_{trans}, respectively. It is important to note that all concepts used in MySQL are part of the ANSI SQL standard and can therefore be applied to any ANSI SQL standard supporting implementation.

A. SQL to canonical

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

$$\begin{aligned}\text{Database} &\Rightarrow \text{Collection}\\ \text{Table} &\Rightarrow \text{Entity}\\ \text{Column} &\Rightarrow \text{Attribute}\\ \text{Foreign keys} &\Rightarrow \text{Relationships}\end{aligned}$$

The first three structures are trivial: a database is a collection of tables and thus entities. The tables have columns, which are represented by the attributes of entities. Relationships between tables in SQL and between entities in our canonical model are however more complicated. 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. For each type of relationship the use of foreign keys are detailed below:

- **One-to-One:** a relationship where a record of a table is connected to at most one record of the other table. In MySQL this is usually defined be two tables having the same primary key. In one of the tables, this primary key is also the foreign key referring to the primary key of the other table. For example, a table "Customer" has a one-to-one relationship with the table "Address". The primary key of "Address" is also its foreign key and refers to the primary key of "Customer". Both tables thus have the same primary key. Another possibility is to have a foreign key in both tables referring to the other table's primary key. One-to-one relationships can also be used for the inheritance between tables, but this specific kind of relationship was not considered at this point.
- **Many-to-One:** a relationship where one record of a table can be connected to multiple records of another table, while the latter are only used in one relation at most. To express this relationship, a foreign key is used in the records on the "many"-side. An example is the relationship between a table "Order" and a table "Customer" where one customer can have many orders, but an order is only related to one customer. In the "Order" table therefore a foreign key is held, referring to the primary key of "Customer".
- **Many-to-Many:** a relationship where records from both tables can be in multiple relations between each other. In MySQL it is not possible to express this relationship with only foreign keys. A new table is therefore introduced

with only two foreign keys mapping records of the two tables, sometimes identified by a single primary key if needed. This map-table has a many-to-one relationship with each of the other two tables. For example, the relation between a table ”Orders” and a table ”Products”. Orders can contain several products, but products can also be in more than one order.

When a foreign key in a table is recognized during the transformation to the canonical model, without any additional information the only correct assumption that can be made is that there exists some kind of relationship with this other table. Therefore, in a first step this general relationship will be translated into the canonical model and, after all tables have been mapped onto entities, the specific types of relationships can be defined in the canonical model as follows:

- **A one-to-one relationship** is thus detected if the local attribute in the entity is also the primary key of that entity or if the related entity also has a relationship referencing the entity.
- **The many-to-many relationship** is more complicated because of an additional table (i.e. entity) introduced by MySQL. This map entity is only used as an aid to represent the many-to-many relationship between two ”true” entities and therefore it will be referred to as a ”false” entity. As mentioned before, it is reasonable to assume this false entity only has two attributes, representing the many-to-one relationships with the two true entities, with the exception of a possible extra attribute serving as an id. The false entity is now marked for deletion, but not effectively removed as data stored in the original table still needs to be transformed afterwards. Finally, a many-to-many relationship is added to both true entities.
- **Many-to-one relationships** are the relationships that remain and do not satisfy the previous conditions.

The entire datastore schema has now been transformed into the proposed canonical model.

B. Canonical to Cassandra

The following schema shows how the canonical data structures are mapped on Cassandra:

$$\begin{aligned} \text{Collection} &\Rightarrow \text{Keyspace} \\ \text{Entity} &\Rightarrow \text{Column family} \\ &\Rightarrow \text{Composite column} \\ &\Rightarrow \text{Super column} \\ \text{Attribute} &\Rightarrow \text{Column} \\ \text{Relationships} &\Rightarrow \ldots \end{aligned}$$

The analogy between collection and keyspace on the one hand, and attribute and column on the other, is straightforward. For the entities and their relationships this is less so. An entity can be any of the following data structures: a column family, a composite column and a super column. Although super columns are no longer favoured as a result of their bad performance, they are only mentioned here for completeness. While relationships are not enforced in Cassandra, they can be represented when present in the canonical model:

- All (true) entities are temporarily considered to be column families
- **One-to-one relationships** are eliminated through inclusion of one entity in the other as a composite column. Which entity is included in the other is decided as follows:
  1) If one of the entities still has other relationships, or more relationships compared to the other, this entity includes the other.
  2) If both entities have no or the same amount of relationships, the entity with most attributes includes the one with less attributes.
  3) If both entities have the same amount of attributes, one of the entities is randomly chosen to be included in the other.
- **Many-to-one relationships** are represented by adding an additional column family to the keyspace. This so called ”index” column family maps the ”one”-side column family on the ”many”-side column family. For example, the column family ”Order” has a many-to-one relationship with the column family ”Customer”. It is easy to determine the customer related to a certain order as this is saved in the column family. A harder query would be to get all the orders for a certain customer. As Cassandra strives towards fast lookup times, joining column families is not an option and therefore an index column family is added allowing these kind of fast lookups. The first consequence of this approach is that more writes are needed to add a record to the ”many” side column family (e.g. ”Order”), but Cassandra is optimized to handle these concurrent writes [8]. Secondly, the datastore is denormalized and data is redundantly stored, which is important to remember when querying or updating the store.
- **Many-to-many relationships** can be similarly addressed as the many-to-one relationships, but from both sides. Therefore two index column families are created to again ensure a fast lookup time. For example, the column families ”Order” and ”Products” with a many-to-many relationship. It is important to have easy access to all the products that are related to an order, but also to all the orders where a specific product is included.

The entire datastore schema has been transformed into a Cassandra datastore and now the data can be transformed from D_{src} to D_{trans} based on the created canonical model. This is done in a similar fashion by mapping the data from MySQL onto the canonical model and then transforming it from the canonical model to Cassandra. The speed layer also performs this transformation in parallel to the batch layer for all the INSERT or UPDATE queries that have been received during the transformation of the structure.

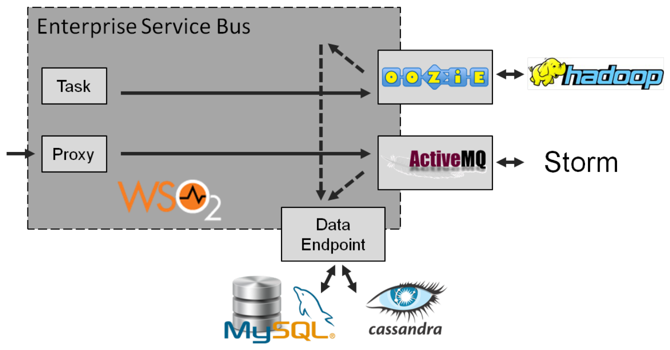

Fig. 3. Instantiation of the framework with all the implemented technologies.

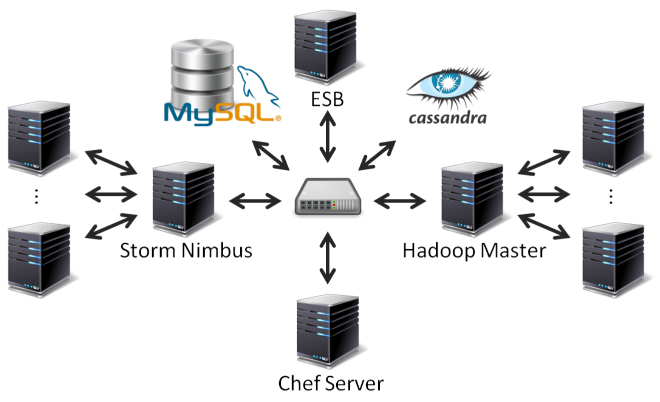

Fig. 4. Setup of the implementation on the iLab.t Virtual Wall.

V. Implementation Details

A. Technology choice and motivation

The proposed architecture in Section II requires an intelligent controller-like component responsible for the communication between the batch/speed layer and handling input/output for those respective layers. Several possibilities were considered, such as a Message Broker (MB), an Enterprise Service Bus (ESB) and a Complex Event Processor (CEP).

A CEP takes actions based on certain events that occur in the system, while a MB allows for the asynchronous communication between applications. The ESB also allows for the communication between applications, but takes a routing approach based on a bus architecture. The CEP may not be as suitable for the proposed architecture as we would have limited control over the messages sent and received. While a MB may suffice for the simple exchange of information between several applications, an ESB allows for more control on the routing, mediation and transformation of the processed messages.

Based on these considerations, the ESB was chosen as central component. The most-used open source ESBs are UltraESB, WSO2 ESB, Mule ESB and Talend ESB. All have an active community with sufficient documentation provided for new users. Performance testing of these open source ESBs shows that on average both the WSO2 ESB and UltraESB have the best performance compared to Mule and Talend [9], [10], [11]. However, the UltraESB is less mature than the WSO2 ESB, having less iterations, and therefore the WSO2 ESB was chosen for the implementation.

A choice also needs to be made regarding the batch layer technologies. In order to achieve such a transformation of a big data set, powerful computing frameworks are needed. One of the best-known batch frameworks is MapReduce [12], originally developed by Google, but made popular by its open-source implementation Apache Hadoop. Another increasingly popular batch framework is Spark [13]. Spark is proven to execute certain programs up to 100 times faster than Hadoop in memory or 10 times on disk. MapReduce is also not the only approach to Big Data analysis. Solutions like the HPCC Systems platform and PowerGraph leverage other programming models to achieve this. However, considering the proposed transformation, there is a distinct similarity with the MapReduce model. The transformation to the canonical model can be considered as a map task, while the conversion from the canonical model can be seen as a reduce task. Based on these findings the batch layer in this proof-of-concept was implemented in Hadoop MapReduce. A scalable workflow management system, Oozie [14], was also installed on top of Hadoop. Oozie also provides a REST API which can be used by the ESB, allowing it to send commands to the Hadoop cluster.

For the speed layer, the most notable candidates are Storm and S4 [15]. Storm, recently introduced in the Apache Incubator project, provides a continuously running topology made up of singular nodes, called bolts, thus creating custom analysis streams. S4 was released by Yahoo in 2010 and also became an Apache Incubator project in 2011. It consists of processing elements, interconnected by streams and bundled in apps. These apps are then deployed and run on nodes. This and a publish/subscribe system of messages makes the framework modular in such a way that apps can interconnect and be assembled in larger systems. Based on the ability of Storm to guarantee processing of the queries and the active community surrounding the project, it was chosen for the proof-of-concept implementation. Both Hadoop and Storm also use Java, which means code is reusable accross both layers. The connection between the ESB and Storm is handled by Java Message Service (JMS) and ActiveMQ [16]. An overview of all the chosen technologies can be found in Figure 3.

VI. Experimental Setup

The implemented instantiation of the architecture was deployed on the Virtual Wall. The iLab.t Virtual Wall facility [1] is a generic test environment for advanced network, distributed software and service evaluation, and supports scalability research. The Virtual Wall contains 100 nodes with Dual CPU (Quad core) with 12GB of RAM and 1x160GB disk.

Figure 4 details the deployment of our implementation on the Virtual Wall. The setup of every node is done through a combination of the JFed software [2] and the configuration management tool Chef [17]. Once the experiment is deployed,

[1]http://ilabt.iminds.be/
[2]http://jfed.iminds.be/

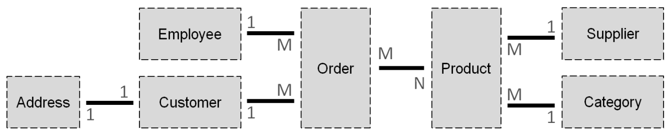

Fig. 5. Structure of the proof-of-concept datastore.

scripts are started on all nodes for the installation of Chef. One node is installed with a Chef server, while all other nodes are installed as a Chef client. Through cookbooks and recipes on the Chef server, the clients are then put into their roles of ESB, Hadoop master, Hadoop slaves, Storm nimbus, and Storm slaves. Oozie is installed on the Hadoop master, while ActiveMQ is installed on the ESB node. Although a choice was made to use Hadoop and Storm for the batch and speed layer respectively, the ESB can support different technologies and a new test environment can be deployed swiftly using JFed, and Chef and its cookbooks.

The structure of the proof-of-concept datastore needs to show the transformation can handle the different types of relationships between entities. With this in mind the structure in Figure 5 is created to serve as a proof-of-concept datastore. It shows a datastore with information concerning a company selling products, provided by a supplier and part of a category. Orders map these products to customers and are handled by an employee. The address of a customer is saved in a separate table.

The original dataset contains 50 tuples in each of the following tables: ”Address”, ”Category”, and ”Customer”. Tables ”Employee” and ”Product” both contain 100 tuples, while tables ”Order” and ”Supplier” contain 200 and 10 tuples, respectively. The table that maps the many-to-many relationship between ”Order” and ”Product” saves 300 tuples. This original setup is then extended linearly to match datasizes of 5, 10, 15, and 20 Gigabytes (GB), where 1 GB of tuples equals 11.5 million tuples. A 20 GB file is therefore equivalent to over 230 million tuples.

VII. Results

A. Batch layer

The batch layer is responsible for the transformation of the structure of D_{src} to D_{trans} and the data contained in the snapshot. Table I details the average execution times for the transformation of the structure of D_{src} in Hadoop. The first job contains the map and reduce function responsible for the initial transformation from MySQL to the canonical model and the optimization step (cfr. Section V). In the second job, the reduce step transforms the canonical representation into Cassandra. Note that the execution times of map and reduce do not add up to the total of each job because of overhead introduced by Hadoop for sorting and routing the data. The execution time of both jobs is also not dependent on the size of the dataset as the structure remains the same.

Figure 6 shows the average execution times for the transformation of the data in Hadoop for increasing dataset sizes. As

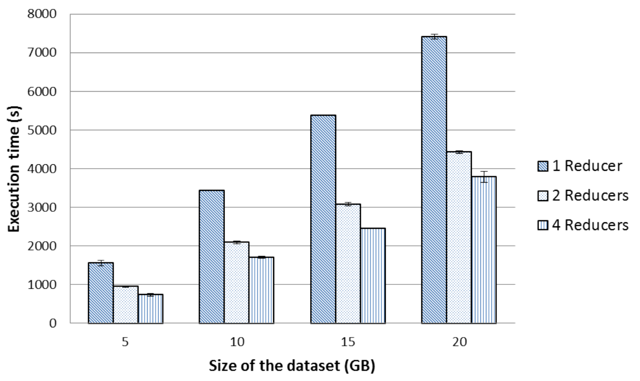

Fig. 6. Average execution times and standard deviation for the transformation of the data of the datastore in Hadoop for increasing dataset sizes.

expected, increasing the dataset size also increases the time needed to transform the data. This increase follows a linear trend. In Hadoop it is also possible to configure the number of parallel reduce tasks in a job. Increasing the number of parallel reducers decreases the execution time, however doubling the number of reducers from 1 to 2 does not halve the execution time. The time gain is even less when increasing the reducers from 2 to 4. This is the result of a Hadoop overhead as it needs to route and copy the data to the correct nodes of the cluster. This becomes more complicated when more reducers are used, and thus the time gain is lowered. The number of map tasks can not be configured directly as Hadoop divides large files in smaller chunks automatically to provide them to a mapper and thus largely decides this autonomously. The number of parallel map tasks in these tests is 4.

B. Speed layer

The speed layer, implemented in Storm, is responsible for transforming INSERT and UPDATE queries for D_{src} into queries for D_{trans}. While we mention INSERT and UPDATE queries as specific query types in ANSI SQL, it is clear that this translates to any query type that inserts new or updates data in any datastore. This process is identical to the data transformation in the batch layer, but with the map and reduce functions accommodated in bolts. Figure 7 depicts the total time for the Storm topology to process the entire set of queries for different query set sizes. In a first stage, every bolt in the topology is limited to 1 executor/task, i.e. no parallel execution of the bolts. A linear increase of the query set yields a linear increase in execution time. This linear trend is confirmed when observing the average overhead for one query passing through

TABLE I
Average execution times for the transformation of the structure of the datastore in Hadoop.

	Job 1			Job 2		
	Map	Reduce	Total	Map	Reduce	Total
Avg (s)	1.1818	7.0909	14.1818	1.3636	8.2727	15.2727
Std dev	0.6030	0.3015	0.7508	0.5045	0.4671	0.4671

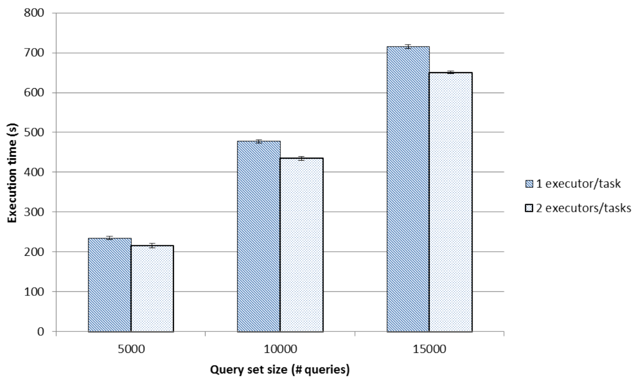

Fig. 7. Average execution times and standard deviation for the transformation of an entire query set in Storm for different query set sizes.

the entire Storm topology in Table II. The average processing time of a query remains constant at around 52ms for increasing query set sizes.

In a second test the number of executors and tasks per bolt was doubled, allowing for parallelism in the bolts. The total execution time in Figure 7 set shows a small gain for all query set sizes. As with Hadoop, Storm also accounts for some overhead for routing the queries through the topology. In Storm, it is also possible to highly tune the parallelism of every bolt in the topology independently. This is necessary because of the varying tasks each bolt has to perform. Execution times may vary between bolts and might lead to bottlenecks in the topology. Simply doubling the capacity of each bolt may therefore not halve the execution time. The results in Figure 7 can thus be seen as an upper limit for the execution times. The average processing time per query remains unchanged (cfr. Table II).

C. Discussion

While increasing the number of reducers in the batch layer yields a decreasing execution time until a certain point, a direct approach will almost always be faster as there is no additional transformation to and from a canonical model. However, this approach was chosen for the needed extensibility of the platform in the heterogeneous storage environment. Additionally, at this point in the transformation workflow, after the batch layer transformed the schema of D_{src}, the speed layer is running in parallel to catch up D_{trans} to the most recent state of D_{src}. The possible negative influence of this approach on the live application(s) is also limited as D_{src} is still the main datastore for all reads and writes during these transformations and the only additional stress on D_{src} will have been the moment where a snapshot of the datastore was taken.

After the transformation, and handover, the speed layer may also be responsible for the continuous transformation of queries, including search and range queries. Although this scenario is outside of the scope of this paper, it is part of the next step towards a system where application changes are no longer required. The results, shown in Figure 7, bode well in this regard with a limited overhead of around 52ms per query.

TABLE II
AVERAGE PROCESSING TIMES PER QUERY IN THE STORM TOPOLOGY.

		1 executor/task	2 executors/tasks
5000	Avg (ms)	52.170	51.817
	Std dev	0.140	0.093
10000	Avg (ms)	51.997	51.705
	Std dev	0.119	0.078
15000	Avg (ms)	51.979	51.670
	Std dev	0.137	0.059

VIII. RELATED WORK

This section discusses related work in the field of migration and transformation of datasets.

The Extract Transform Load (ETL) principle is a process, frequently used in data warehousing, where data is extracted from an outside source, transformed according to several rules into a predefined format and loaded into an operational datastore or data warehouse. The proposed framework can be considered as a type of ETL framework: the structure and data are extracted from a source datastore, they are transformed and loaded into a new datastore. While ETL processes focus on data alone and are often part of a long term solution [18], [19], the proposed framework transforms both structure and data and loads it into a newly created datastore instead of an already operational store. However, as the process shows a large resemblance with any data transformation, this section follows the different steps in the process, i.e. extract, transform and load.

Before any transformation can be performed, the schema and data of D_{src} needs to be extracted and migrated to the framework. The migration of big data sets has already been researched thoroughly. The obstacles posed by migration have been approached in numerous ways, such as using high-performance networks [20], having a workload-aware strategy [21], or through a cost-minimizing approach [22]. Considering the cloud context, an additional obstacle arises as many applications have to meet strict service-level agreements (SLA), therefore the downtime of the applications needs to be limited or eliminated entirely. Performing a data migration with no downtime of the application is called a *live* migration. With the growing popularity of the cloud, extensive research has been done in this sub-domain of data migration [21], [23], [24]. The Albatross technique for shared storage, developed by Das et al. [23], uses an iterative technique where a snapshot of the source datastore at the destination tries to catch up by iteratively copying the changes. Elmore et al. propose a technique for shared nothing datastores where pages of the store are pulled on-demand by the destination [24]. Both proposals do however assume several characteristics of their respective datastores. As this framework aims to support any datastore and therefore to be easily extensible, assuming anything about possible datastores would limit the flexibility of the framework significantly. While they perform no or limited changes to a datastore's structure or data, they also provide some interesting insights into the ”live” aspect for the proposed framework.

A domain directly related to transformation of schemas and data, is schema matching and mapping [25]. Schema matching is the task of finding semantic correspondences between elements of two datastore schemas, while schema mapping aims to find a query or set of queries to map a source datastore into a destination datastore. Challenges in the automation of this process have been largely related to the heterogeneity, e.g. different technologies or semantics. The advent of NoSQL datastores has not eased these issues due to their own heterogeneity and flexible, or even schemaless, datamodels. Ontologies have provided a solution to the semantic heterogeneity in the form of ontology matching [26], but many other challenges exist. While this entire domain provides many solutions for transforming data, it differs from the problem this paper solves in the sense that both the data schemas are known in advance in schema matching and mapping. In this framework the schema of D_{trans} is not known in advance, but is derived from the information in D_{src}.

IX. Conclusion

This paper proposes an approach for the live transformation of datastores through a canonical model. A new framework is introduced, based on the concept of the Lambda architecture with a parallel batch and speed layer. The framework is implemented as an Enterprise Service Bus with Hadoop and Storm as services for the batch and speed layer, respectively. This prototype showcases an implementation of the transformation between a MySQL database and a Cassandra NoSQL store. Results show a linear trend in execution times for increasing dataset sizes in the batch layer. Hadoop overhead also limits the time gain when increasing the number of parallel reducers. In the speed layer, Storm allows for a quick catch-up by D_{trans} after the batch layer has finished it's schema transformation and before the changeover from D_{src}. These results are also promising for a situation where a continuous transformation is necessary to avoid the application changes. While the impact of this continuous transformation needs to be evaluated further, these initial results prove that the speed layer is able to limit the introduced overhead.

Other future work will focus on the integration of this new framework into the Tengu platform for dynamic changeovers between data models. Here, monitoring data will be used to identify turning points in the performance of applications using specific datastores in order to autonomously decide when a transformation is needed.

Acknowledgment

This work has partly been funded by the IWT TWIRL project (110580) and the iMinds (DMS)^2 project (120442).

References

[1] M. J. Skok, "The 3rd Annual Future of Cloud Computing," tech. rep., North Bridge and GigaOM, 2013. http://goo.gl/mDmkst.

[2] A. Li, X. Yang, S. Kandula, and M. Zhang, "Cloudcmp: comparing public cloud providers," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pp. 1–14, ACM, 2010.

[3] A. Ruiz-Alvarez and M. Humphrey, "An automated approach to cloud storage service selection," in *Proceedings of the 2Nd International Workshop on Scientific Cloud Computing*, ScienceCloud '11, (New York, NY, USA), pp. 39–48, ACM, 2011.

[4] A. Jacobs, "The pathologies of big data," *Commun. ACM*, vol. 52, pp. 36–44, Aug. 2009.

[5] T. Vanhove, J. Vandensteen, G. Van Seghbroeck, T. Wauters, and F. De Turck, "Kameleo: Design of a new Platform-as-a-Service for Flexible Data Management," in *Proceedings of the 2014 IEEE/IFIP Network Operations and Management Symposium (NOMS 2014)*, 2014.

[6] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable realtime data systems*. Greenwich, CT, USA: Manning Publications Co., 2014. (Early Access Program).

[7] P. P.-S. Chen, "The Entity-relationship Model - Toward a Unified View of Data," *ACM Trans. Database Syst.*, vol. 1, pp. 9–36, Mar. 1976.

[8] P. McFadin, "The Data Model is dead, long live the Data Model," DataStax Webinar, may 2013.

[9] D. Abeyruwan, "ESB Performance Round 6.5," tech. rep., WSO2, jan 2013. http://wso2.com/library/articles/2013/01/esb-performance-65/.

[10] A. C. Perera and R. Linton, "ESB Performance Round 7," tech. rep., AdroitLogic, oct 2013. http://esbperformance.org/display/comparison/ESB+Performance.

[11] S. Anfar, "ESB Performance Round 7.5," tech. rep., WSO2, feb 2014. http://wso2.com/library/articles/2014/02/esb-performance-round-7.5/.

[12] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[14] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur, "Oozie: towards a scalable workflow management system for hadoop," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, p. 4, ACM, 2012.

[15] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pp. 170–177, dec 2010.

[16] B. Snyder, D. Bosnanac, and R. Davies, *ActiveMQ in action*. Manning, 2011.

[17] M. Marschall, *Chef Infrastructure Automation Cookbook*. Packt Publishing, 2013.

[18] S. Henry, S. Hoon, M. Hwang, D. Lee, and M. D. DeVore, "Engineering trade study: extract, transform, load tools for data migration," in *Systems and Information Engineering Design Symposium, 2005 IEEE*, pp. 1–8, IEEE, 2005.

[19] H. Agrawal, G. Chafle, S. Goyal, S. Mittal, and S. Mukherjea, "An enhanced extract-transform-load system for migrating data in telecom billing," in *IEEE 24th International Conference on Data Engineering (ICDE 2008)*, pp. 1277–1286, IEEE, 2008.

[20] B. W. Settlemyer, J. D. Dobson, S. W. Hodson, J. A. Kuehn, S. W. Poole, and T. M. Ruwart, "A technique for moving large data sets over high-performance long distance networks," in *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, MSST '11, (Washington, DC, USA), pp. 1–6, IEEE Computer Society, 2011.

[21] J. Zheng, T. S. E. Ng, and K. Sripanidkulchai, "Workload-aware live storage migration for clouds," *SIGPLAN Not.*, vol. 46, pp. 133–144, Mar. 2011.

[22] L. Zhang, C. Wu, Z. Li, C. Guo, M. Chen, and F. Lau, "Moving big data to the cloud: An online cost-minimizing approach," *Selected Areas in Communications, IEEE Journal on*, vol. 31, pp. 2710–2721, dec 2013.

[23] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration," *Proc. VLDB Endow.*, vol. 4, pp. 494–505, May 2011.

[24] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: Live migration in shared nothing databases for elastic cloud platforms," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, (New York, NY, USA), pp. 301–312, ACM, 2011.

[25] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *the VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[26] J. Euzenat and P. Shvaiko, *Ontology Matching*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.