2016 IEEE International Conference on Software Maintenance and Evolution

Hug the Elephant: Migrating a Legacy Data Analytics Application to Hadoop Ecosystem

Feng Zhu[†*], Jie Liu[†¶], Sa Wang[‡], Jiwei Xu[†], Lijie Xu[†], Jixin Ren[§], Dan Ye[†], Jun Wei[†¶] and Tao Huang[†¶]
[*]University of Chinese Academy of Sciences
[¶]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[†]Institute of Software, Chinese Academy of Sciences
[‡]Institute of Computing, Chinese Academy of Sciences
[§]Xin Yi Hua Medical Technology Company, Zhengzhou, Henan
{zhufeng10, ljie, xujiwei, xulijie09, yedan, wj, tao}@otcaix.iscas.ac.cn, renjixin@163.com, wangsa@ict.ac.cn

***Abstract*—Big data applications that rely on relational databases gradually expose limitations on scalability and performance. In recent years, Hadoop ecosystem has been widely adopted as an evolving solution. This paper presents the migration of a legacy data analytics application in a provincial data center. The target platform follows "no one size fits all" method. Considering different workloads, data storage is hybrid with distributed file system (HDFS) and distributed NoSQL database.**

Beyond the architecture re-design, we focus on the problem of data model transformation from relational database to NoSQL database. We propose a query-aware approach to free developers from tedious manual work. The approach generates query-specific views (*NoView*) for NoSQL and re-structures the views to align with NoSQL's data model. Our results show that the migrated application achieves high scalability and high performance. We believe that our practice provides valuable insights (such as NoSQL data modeling methodology), and the techniques can be easily applied to other similar migrations.

***Keywords*—Hadoop, Migration, Data Model, NoSQL Database**

I. Introduction

In the big data era, a large number of organizations are confronted with challenges on software evolution, which is typically driven by the ever-increasing data size. Particularly, for data-intensive applications, the 5V (volume, velocity, variety, veracity, and value) [1] characteristics promote the demands on excellent storage and processing capabilities beyond traditional architectures. A common and evolving strategy is to migrate the application to more modern platforms. As the *de facto* standard for big data techniques, Hadoop[1] ecosystem has become the very choice in a variety of scenarios. Consequently, many enterprises migrate their legacy applications (e.g., [5], [6], [7], [8] and etc.) to Hadoop ecosystem for scalability, performance and flexibility through clusters of commodity hardware.

Our research centers on big data analytics applications, which are generally built to collect, store, process and query data. While modern companies take their data as a valuable asset, big data analytics for mining this asset in particular has been regarded as the key discipline in the last decade [37]. Hence, it is common in various domains (e.g., social network, healthcare, online shopping and etc.) to uncover the hidden patterns and get statistical information for business intelligence, government policy support and so on.

However, migrating to Hadoop ecosystem is a non-trivial task for developers. Challenges for conducting such a migration project are manifold. To begin with, traditional solutions based on relational database or data warehouse are so-called "one size fits all" [9], [32]. Nevertheless, the major trend in big data is the understanding that there is "**no one size fits all**" solution [9], [10], [11], [12]. In relational databases, data storage and query processing are tightly-coupled as a whole. When migrating to Hadoop ecosystem, these functionalities need to be provided by different decoupled frameworks, like HDFS, NoSQL [4], MapReduce [3], and so on. The wide diversification of features and interfaces requires much effort to make them work together well. Second, architecture evolution brings challenges in data migration process. Prominently, data model layer will correspondingly see an up-to-down shift between the source and target platform, varying from data modeling methodology to physical storage structure, resulting to the difficulties in the transformation process. However, there is no general guideline to assist developers.

The objective of this paper is to demonstrate the methods, approaches and techniques to address the above challenges in the context of data analytics applications. We carry out a detailed case-study to (1) adapt a repeatedly-used migration process model, (2) re-design the architecture based on Hadoop ecosystem, and (3) present an automatic query-aware approach for the data model transformation problem, which is embodied in data migration. The contributions of this paper can be summarized as follows.

- The successful case provides feasible guidelines in architecture and dataflow design for similar migrations to Hadoop environment, which is a trend in the big data era.
- We focus on the problem of data model transformation (from relational database to NoSQL database). To the best of our knowledge, this is the first paper with an available automatic approach to address the problem in the context of data analytics applications.
- Techniques and patterns proposed in our work are general and application-independent. We conduct an in-depth study to give insights and reveal the generality.

[1]Apache Hadoop. http://hadoop.apache.org

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.14

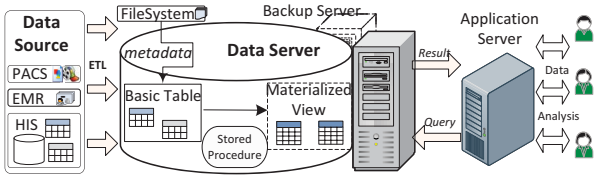


Fig. 1. The Initial Architecture of HEALTH Application

The rest of this paper is organized as follows. Section II describes the application we studied in this paper. In section III, we present the migration process and the new architecture based on Hadoop ecosystem. Section IV elaborates on the problem of data model transformation. In section V, we evaluate the migration and share lessons learned. Related work is given in section VI. Section VII makes the discussion on generality and threats to validity. We conclude this paper and our future work in section VIII.

II. Background: the Application

Data analytics applications are common and significant in many organizations. Since the summer of 2013, we have been engaged in such an application for healthcare in Henan province (China), called HEALTH[2], which technically supports the national "new village cooperative health insurance" project. HEALTH was first developed in 2009 to provide insights in medical data, including structured data (relational tables), semi-structured data (medical documents) and unstructured data (medical images, videos).

Fig. 1 depicts the architecture of HEALTH. The central data server combines data residing in disparate sources. In current stage, structured relational data analysis is the main task. Medical documents, images and videos are stored in filesystem with their metadata (i.e., *path*, *keyword*, and etc.) maintained in database. The data center creates basic tables the same as that in data sources. New records for the basic tables are loaded periodically by an open-source ETL (Extraction-Transformation-Load) tool Kettle[3]. Then, Kettle calls the designed stored procedures to incrementally update the pre-defined materialized views [14]. HEALTH provides services as parametric queries. In the query time, users input particular parameters through the presentation layer (such as web textarea, drop-down list, etc.) to get result by performing SQL queries on basic tables and materialized views.

Workload Types. HEALTH does not have data updates or deletions. The read-only workloads can be divided into two types: *interactive query* and *batch-oriented reporting query* [15]. It should be noted that, there is no strict boundary between these two types in technical side. In HEALTH, the classification is marked clearly according to different scenarios, which can be recognized as below.

- *Interactive Query.* Interactive queries need to guarantee response time to meet users' online waiting for the results, and cost from milliseconds to several minutes. The queries are generally simple without not too many sub-queries.
- *Batch-oriented Reporting Query.* Reporting queries deliver off-line business reports as results to users with comprehensive analysis in a batch way, and cost tens minutes or even up to hours. The queries are complex with substantial sub-queries, aggregation and join operations.

This initial architecture has worked well for nearly four years. With the ever-increasing data size, the architecture gradually exposes limitations. (1) The first and most obvious one is the difficulty for scaling out. Until 2013, HEALTH had covered more than 150 counties, with 26TB accumulated data and 28GB newly powered data every day. Relying on relational database server is difficult to achieve the scalability. (2) The other drawback is the performance bottleneck. The maintenance time between basic tables and materialized views keeps increasing and some queries even take up to ten hours. Though techniques like table partition can be made to alleviate the pressure, data accumulation leads to the optimization endless. (3) Moreover, considerations should anticipate the future demands. HEALTH has now started the plan on big data mining and machine learning (e.g., the disease prediction). To conclude, we need an evolving architecture that not only meets the current requirements, but also anticipates the future extensions. Motivated by the facts above, we decided to migrate the HEALTH application to Hadoop ecosystem.

III. Migration Process and Architecture Design

A. Preliminary: Hadoop Ecosystem

Hadoop is the open source implementation of Google's distributed file system GFS [2] and parallel computing framework MapReduce [3]. Since it emerged in the early 2000s, a rich ecosystem has been developed and gained its popularity. Until now, the new Hadoop 2.0 ecosystem has included various components, such as Apache Spark[4], Hadoop's distributed file system (HDFS), NoSQL [4] databases and so on. Besides the basic components of HDFS and MapReduce in Hadoop, the new architecture of HEALTH is based on three well-known frameworks: Hive[5], HBase[6] and Sqoop[7].

Hive [16] is the data warehouse solution beyond Hadoop. It stores data as relational tables in Hadoop's distributed filesystem (HDFS) and provides a SQL dialect (HiveQL) to express queries on tables. A Hive query will be converted to a sequence of MapReduce jobs.

HBase is the distributed NoSQL database built on the top of HDFS, modeling on Google's BigTable [17]. HBase not only supports random, real-time data read and write access, but also can take advantage of MapReduce for batch-processing tasks.

Sqoop is a ETL tool designed to transfer data between Hadoop and relational databases. The dataset being transferred is sliced up into different partitions and a MapReduce job is

[2]HEALTH in Henan Province. http://www.hnhzyl.com
[3]Kettle ETL Tool. http://kettle.pentaho.org
[4]Apache Spark. http://spark.apache.org
[5]Apache Hive. http://hive.apache.org
[6]Apache HBase. http://hbase.apache.org
[7]Apache Sqoop. http://sqoop.apache.org

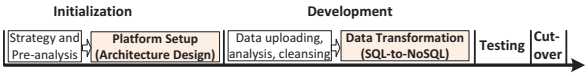


Fig. 2. Four-step Data Migration Process Model

launched with individual mappers responsible for transferring a slice of the dataset.

B. Migration Process Model

The migration of HEALTH revolves around data. However, any unplanned movement in shape of unprofessional data migration leads to high risk. A stringent and stepwise approach is critical. In our work, we adapted a mature practice-based data migration process model [18], shown in Fig. 2. The process model consists of four main stages, which in turn contain distinct phases. The main stages are: initialization, development, testing and cut-over.

Initialization. Initialization is the preparing stage before starting the migration. The project organization and technical infrastructure for HEALTH are established. The stage contains phases for strategy and pre-analysis (e.g., scope, roadmap, risks estimation and etc.), and the platform setup.

In Fig. 2, we highlight the architecture re-design, which is the key problem in platform setup phase. As aforementioned, the target platform based on Hadoop ecosystem follows a "no one size fits all" method. Considering the different query types, the new architecture will adopt a **hybrid** storage to combine the benefits of different storages. For batch-oriented reporting queries, data is kept in HDFS and managed by Hive. The batch-oriented Hive (on HDFS) is appropriate for reporting queries that involve intensive whole table scans to produce reports. To support interactive queries, some specific data is generated to be re-structured in HBase.

Development. The development stage covers all aspects for implementing the migration program. It is vital to learn as much as possible about the data and its structure of both source and target platform. As a matter of fact, the development stage consists of two distinct phases on data uploading, analysis, cleansing, and data transformation. In Fig. 2, we highlight the data transformation phase, which is performed as an incremental and iterative manner.

Incremental and iterative manner. To mitigate data migration risks, the whole data transformation process is incremental and iterative. (**i**) The first step is straightforward with one-to-one mapping for all queries: one relational basic table to one Hive basic table, one materialized view to one intermediate table (as no concept for "materialized view" in Hive), and one SQL query to one Hive SQL query. (**ii**) Then for interactive queries, we will consider how to implement the data storage and data access patterns in HBase.

SQL-to-NoSQL. The data stored in HBase raises the problem of data model transformation from relational database to NoSQL database (i.e., SQL-to-NoSQL). The unique characteristics (i.e., data modeling methodology and flexible structure) of NoSQL make the problem challenging and difficult. On one hand, the schema-less design of NoSQL indicates numerous mapping schemes. On the other hand, relational database is optimized with sophisticated mechanisms like index and cost-based query engine. When migrating to NoSQL, these features cannot be inherently supported but their advantages for performance need to be achieved. Nevertheless, there is no general guideline to assist developers until now.

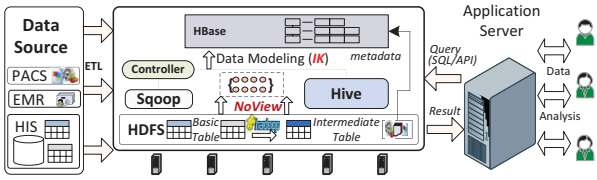


Fig. 3. The Architecture of HEALTH based on Hadoop Ecosystem

To handle the problem, we propose an automatic query-aware approach. Given a query and relational tables, our approach accomplishes the transformation by two phases: (1) generates the query-specific view for NoSQL by analyzing query's abstract syntax tree (*AST*), and then (2) reorganizes the attributes to conform NoSQL storage according to the query and indexes created. The first phase is in fact the reverse engineering on legacy relational solution. For presentation convenience, we denote the generated views as *NoView*. The second phase is the process of NoSQL data modeling for *NoView* (i.e., "NoSQL View"). We concentrate on the design for row-key of NoSQL's flexible data model to facilitate data access. We call the design as *IK* (short for "intelligent key").

Testing & Cut-Over. The testing stage validates migration effects on correctness, performance, scalability and so on. In this stage of our migration, solutions of relational database and Hadoop ecosystem are co-existed. In the end, the cut-over stage switches the application to the Hadoop ecosystem.

C. Architecture Redesign

Fig. 3 demonstrates the new architecture based on Hadoop ecosystem. In distributed environment, Hadoop cluster and HBase cluster share the same master node, on which both Sqoop and Hive are installed. The controller is a lightweight coordinator we implemented to schedule different frameworks to work together in order (the initial choice is Oozie[8], but we found it is too complicated). The controller runs as a non-stop service in the master node. It encapsulates a set of scripts that manage the lifecycles (i.e., the *startup* and *shutdown*) of different frameworks. The scheduling logic can be described from three stages in terms of dataflow: data source (importing data), data storage and query processing.

Data Source. For newly produced medical documents, images and videos, we use a MapReduce program to fetch them. For relational tables, Sqoop takes Kettle's place to import new records from data sources in parallel. As that in initial architecture, data importing job is scheduled by the controller in midnight every day.

[8]Apache Oozie Workflow Scheduler for Hadoop. http://oozie.apache.org

Data Storage. The semi-structured data and unstructured data are stored in HDFS and their metadata is managed in HBase. For structured data, the new architecture adopts a hybrid storage solution to combine the batch-processing character of Hive (on HDFS) and the real-time data access feature owned by HBase. The long running queries are performed on basic tables and intermediate tables (corresponding to the materialized views in relational database) in Hive, while the interactive queries are executed on *NoView* in HBase.

New records from different data sources will be imported directly to the basic tables in Hive (on HDFS). These tables are defined as the *partitioned table* in Hive, therefore the incremental data imported everyday will be automatically appended as a new partition table tagged by date time. Then, the controller starts the designed HiveQL (corresponding to the *stored procedure* in relational database) to incrementally update the intermediate tables and *NoView*.

Query Processing. Query processing in HEALTH includes two sides. One is the consistency maintenance between basic tables and derived data (i.e., the intermediate tables and *NoView*). The new architecture employs HiveQL (translating to MapReduce) to accomplish the task in a batch way. The other side is the query services provided to users. For batch-oriented long running queries, data access pattern is expressed as HiveQL. For interactive queries, data access pattern may be HBase API or the hybrid of it with MapReduce.

IV. Data Model Transformation

Data model transformation indicates the method shift from one to another one. In this section, we devote special attentions to the SQL-to-NoSQL problem.

Target Data Model. HBase is one kind of extensible records NoSQL store. The most basic unit of HBase is a *column*. One or more columns form a *row* that is addressed uniquely by a row key. A number of rows form a *table*. All rows are always sorted ascend-lexicographically. Rows are composed of columns and group them into column families. *Column family* builds topical boundaries between the data. Columns are often referenced as *family:qualifier*. HBase mainly provides three data access interfaces: *Get*, *Put* and *Scan*. The *Get* and *Put* interfaces are specific to particular rows and need the row key to be provided. The *Scan* operation is performed over a range of rows defined by a start and stop row key.

A. Running Example and Approach Overview

Fig. 4 shows the running example. Two tables (*os_pres_01* and *os_pres_02*) record information of prescriptions while the *gr_pyc_code* table represents the information of physicians. The *drug_gr_info* is the table with drug attributes. *PT* and *PN* are two interactive queries. The *PN* query is to *"figure out the top k physicians who use penicillin in his prescription during a specific time period"*, while the *PT* query is to *"find out the most recent k physicians who use penicillin"*. The *drug_name* and *pres_time* are parameters reserved for users' query-time inputs. The strategy for materialized view is to join *os_pres_01* and *os_pres_02* on the field *pres_id* to get *os_pres*. For performance improvement, the two columns (*drug_name* and *pres_time*) are indexed.

Fig. 5 depicts the overview of whole data model transformation process. The left part is the solution with Hive (on HDFS), and the right part is the solution based on hybrid storage. It can be seen that the materialized view has been re-designed as *NoView*. The entire process consists of two fundamental components: *NoView* and *IK*.

Before transformation. The solution based on Hive (on HDFS) leads to poor performance for interactive queries. (1) Hive translates queries into a sequence of MapReduce job, which is time-consuming with batch-oriented character. (2) The indexes created in relational database are dropped and consequently query processing will incur the scan on whole table. To fill the gap, we use HBase as a complementary storage to support the interactive queries.

Transformation approach: query-aware. The core method of query-aware approach is to store the reverse engineered *NoView* with fine-grained key-value based modeling according to the query and indexes. The *Noview* can be treated as a special view that is generated from basic tables with HiveQL. The fundamental purposes of these two components are the answers to the questions on "what data should be stored in NoSQL" and "how to store the data".

After transformation. The right part in Fig. 5 shows the data model and data access pattern after transformation. Our approach generates the *NoView* with four attributes {*drug_name*, *pres_time*, *pyc_name*, *drug_number*}. The row key is the composition of *drug_name* and *f(pres_time)*, delimited by the char "+", in which the function *f(dateTime)* reverses the timestamp's lexicographical order. For example, the timestamp string "2012-01-01" will be encoded as "7987-98-98", subtracting from 9 for each number. Each row contains a column *pyc_name* with the cell of *pyc_number*. The rewritten *PT* query scans table with start row key *"penicillin+"*, until getting 5 unique users. The rewritten *PN* query will calculate the top 10 physicians during the range scan between start row key *"A+f(endTime)"* and end row key *"A+f(startTime)"*.

B. NoView: Query-specific View for NoSQL

NoView, short for "NoSQL View", is a special view, which contains the minimum query-specific attributes. There may be numerous strategies to generate some specific data beyond basic tables to support an interactive query. For NoSQL data modeling methodology, *NoView* adopts an extreme way to get optimal query performance. For the running example, only the attributes of {*pyc_name*, *drug_name*, *drug_number*, *pres_time*} are necessary to the *PT* query and *PN* query.

Reasons for *NoView*. To begin with, we define a query is *easy-implementable* to NoSQL if it contains no sub-queries or *join* operations in the context of this paper. We take *NoView* based on the following reasons. (1) For other strategies, if the transformed query on generated data is not *easy-implementable*, they will be difficult to support interactive queries, because the *join* operations lead to intensive data shuffling in NoSQL's distributed environment and the logic

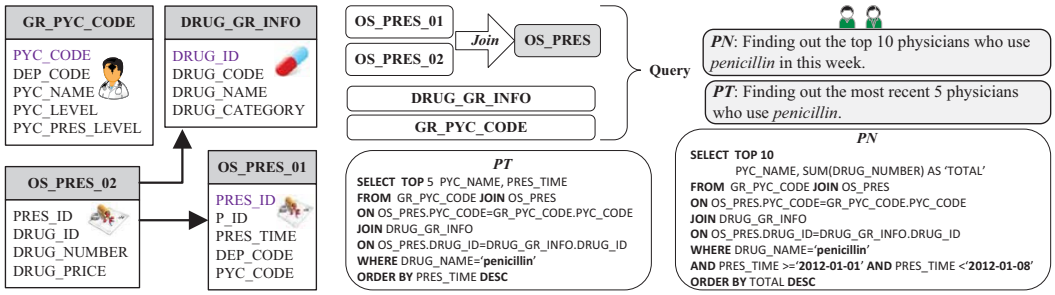


Fig. 4. Running Example for Data Model Transformation

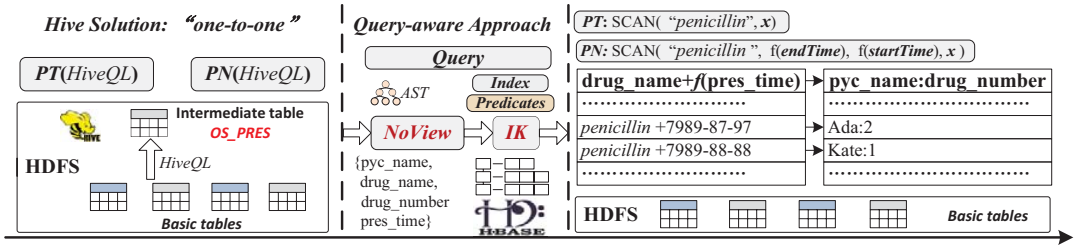


Fig. 5. Data Model Transformation Overview

of sub-queries is complex to be implemented with low-level NoSQL APIs. Hence, they generally have the similar computing costs as that of *Noview*. (2) *NoView* provides a most possible way with maximum pre-computation. If it cannot simplify the initial query to an *easy-implementable* query, other strategies will fail too. (3) Most importantly, *NoView* generation and maintenance are accomplished by MapReduce (translated by Hive) in a batch way. Due to its scalability, the time cost can be ensured within certain time span by adding servers. This is the main difference with traditional single relational database, which requires the tradeoff between two sides (i.e., data maintenance and the query side).

TABLE I
OPERATOR AND QUERY NOTATIONS

Selection: $\sigma_S(R)$	R is the relation table and S is the column set
Projection: $\pi_P(R)$	R is the relation table and P is the column set
Join: $R_1 \bowtie_J R_2$	R_1 and R_2 are relation tables and J is the column set
Aggregation: ${}_A\mathscr{G}_F(R)$	R is the relation table, A is the column set in group-by clause and F is the column set in aggregation
Order By: $\mathscr{O}_{C(a/d)}(R)$	R is the relation table, C is the column set and a/d represents ascending/descending
Top: $\mathscr{T}_K(R)$	R is the relation table and K is top number

***NoView* Generation.** *NoView* is generated by pushing up the parameters with the re-arrangement of query operators. Without the loss of generality, we study the queries with frequent operators: *selection*, *projection*, *join*, *aggregation*, *sort* and *limit*. Based on the notations, the *PT* query can be expressed as: $PT = \mathscr{T}_x\mathscr{O}_{C(d)}(\pi_P(R_1 \bowtie_{J_1} (\sigma_S(R_2 \bowtie_{J_2} R_3))))$, in which $P = \{pyc\_name, pres\_time\}$, $S = \{drug\_name\}$, $J_1 = \{pyc\_code\}$, $J_2 = \{drug\_id\}$, $R_1 = gr\_pyc\_code$, $R_2 = os\_pres$, $R_3 = drug\_gr\_info$ and $C = \{pres\_time\}$.

Considering a parameter in *where* expression followed by two operators, $\mathscr{X}$ and $\mathscr{Y}$, i.e., $\mathscr{X}\mathscr{Y}(\sigma_S(R))$. (1) When $\mathscr{Y}$ is *selection*, *projection*, *join* or *sort*, the parameter can be extracted with $\mathscr{X}\mathscr{Y}(\sigma_S(R)) = \mathscr{X}(\sigma_S\mathscr{Y}(R))$. Furthermore, any valid *PSJ*-expression can be transformed into a standard form consisting of cartesian product, followed by a selection, followed by a projection [19]. (2) When $\mathscr{Y}$ is the *aggregation* operator (i.e., $\mathscr{X}({}_A\mathscr{G}_F(\sigma_S(R)))$) and $\mathscr{X}$ is a *selection*, *projection* or *sort* operator, the expression can be converted into $\mathscr{X}({}_A\mathscr{G}_F(\sigma_S(R))) =_A \mathscr{G}_F(\mathscr{X}(\sigma_S(R)))$. When $\mathscr{X}$ is a *join* operator, column sets J, A, F and R in aggregation have relations as $A \cap F = \emptyset$, $A \cap (R - A - F) = \emptyset$, $F \cap (R - A - F) = \emptyset$. If $S \subseteq A$, aggregation can be computed before the selection, i.e., ${}_A\mathscr{G}_F(\sigma_S(R)) = \sigma_S({}_A\mathscr{G}_F(R))$, so the parameters can still be equally extracted with $\mathscr{X}({}_A\mathscr{G}_F(\sigma_S(R))) = \sigma_S(\mathscr{X}({}_A\mathscr{G}_F(R)))$. Otherwise, the aggregation cannot be pre-computed. In this case, to accomplish the $\mathscr{X}$ operator first can be tried. As for the *join* operator, if $S \not\subseteq A$ and $J \cap F = \emptyset$, then the joining of columns will not change after aggregation and the *join* operator can be pre-computed. Therefore, the below equation can be get $R_1 \bowtie_J ({}_A\mathscr{G}_F(\sigma_S(R))) =_A \mathscr{G}_F(\sigma_S(R_1 \bowtie_J R))$. If $S \not\subseteq A$ and $J \cap F \neq \emptyset$, neither *join* operator can be put inside nor the parameters can be put outside of the *aggregation* operator. (3) When the parameters cannot be pulled out from $\mathscr{Y}$. The generation will be stopped. Therefore, with combing the query that joins *os_pres_01* and

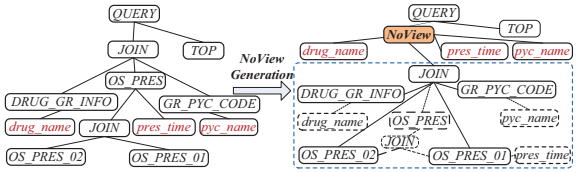


Fig. 6. *NoView* Generation for the *PT* Query

os_pres_02 for table *os_pres*, the *PT* query can be transformed as $PT = \mathscr{T}_x\mathscr{O}_{C(d)}\sigma_S(NoView) = \mathscr{T}_x\mathscr{O}_{C(d)}\sigma_S(\pi_{P_1}(R_1 \bowtie_{J_1} ((\pi_{P_2}(R_4)) \bowtie_{J_3} (\pi_{P_3}(R_5))) \bowtie_{J_2} R_3))$, in which, $P_1 = \{pyc\_name, pres\_time, drug\_name\}$, $P_2 = \{pres\_id, pyc\_code, pres\_time\}$, $P_3 = \{pres\_id, drug\_id\}$, $R_4 = os\_pres\_01$, $J_3 = \{pres\_id\}$, $R_5 = os\_pres\_02$.

The algorithm works at the abstract syntax tree (*AST*) of a parametric query with a bottom-up and depth-first behavior. When coming across the parameters in *where* clause in an expression composed with *selection*, *projection*, *join* and *sort* operators, it pushes them up to the upper layer and keeps the parameters in *select* clause. Fig. 6 shows the process with *PT* query. When coming across the expression with *aggregation* operator, the *join* operator in upper layer will be pushed down to be computed first. When coming across the *aggregation* operator and the parameters cannot be pushed up, the *join* expressions will be pulled in. When the operator is the *limit* or reaching the top layer, the algorithm will end the search, and return the root node of *AST* for sub-tree to generate *NoView*.

***NoView* Maintenance.** We use the notation $\triangle$ to represent the incremental data. As $\sigma_S(R + \triangle R) = \sigma_S(R) + \sigma_S(\triangle R)$, $\pi_P(R + \triangle R) = \pi_P(R) + \pi_P(\triangle R)$, $R_1 \bowtie_J (R_2 + \triangle R_2) = R_1 \bowtie_J R_2 + R_1 \bowtie_J \triangle R_2$, and the *sort* operator can be inherently supported with NoSQL's naturally sorted trait, query with any combination of *selection*, *projection*, *sort* and *join* operators can be processed in incremental way with the same expression as that for *NoView* generation. When the columns for aggregation in incremental data set disjoint with that in *NoView*, $\pi_A(R) \cap \pi_A(\triangle R) = \emptyset$, we get ${}_A\mathscr{G}_F(R + \triangle R) =_A \mathscr{G}_F(R) +_A \mathscr{G}_F(\triangle R)$. Otherwise, the equation is ${}_A\mathscr{G}_F(R+\triangle R) \neq_A \mathscr{G}_F(R)+_A\mathscr{G}_F(\triangle R)$. Therefore, (1) if *NoView* is generated by expressions which contain only fd operators, it can be maintained in an incremental way with the same expression. (2) When all the *aggregation* operators satisfy the above condition, the *NoView* can be incrementally maintained. (3) Or the *NoView* must be re-generated.

***NoView* Reuse.** Different queries may obtain the same or similar *NoView*s. Reusing *NoView* seeks these similarities to save the storage costs. While *NoView* is a special view, the problem can be addressed with Query Graph Model (QGM) [20]. In QGM, a query is represented as a rooted graph with different boxes (nodes). Through comparing conditions and output columns for each two corresponding tree layers from leaf boxes to the root boxes, we can validate whether the *NoView* can be reused. If two candidates are produced by the same box-line, there are two cases for reusing: (1) Exact reuse. They are totally the same data set. (2) Subset reuse. One is the subset of the other one, the corresponding two queries will take the superset one. For instance, as *NoView*(*PT*)={*pres_time*, *pyc_name*, *drug_name*}, *NoView*(*PN*)={*pres_time*, *pyc_name*, *drug_name*, *pyc_number*} and they are generated with the same expression, *NoView*(*PN*) can be reused in the query *PT*.

C. IK: NoSQL data modeling for NoView

The *key* part is the entrance of NoSQL's key-value based world. It plays a critical role in NoSQL data modeling. An intelligent key can implement the index mechanism in relational database and facilitate data access.

Common Design Patterns. To begin with, we summarize several common patterns for intelligent key design in NoSQL.

Composite Key. The composite key puts multiple fields into row key to keep the related records together, as demonstrated in Fig. 7(a). For the query *PN*'s *NoView*, the key composites *drug_name* and *pres_time*. This composite key design avoids unnecessary scan. For example, if the time range is the year of 2014 and the drug name is "*penicillin*", we only need to scan records with the start key "*penicillin_20140101*" and end key "*penicillin_20150101*".

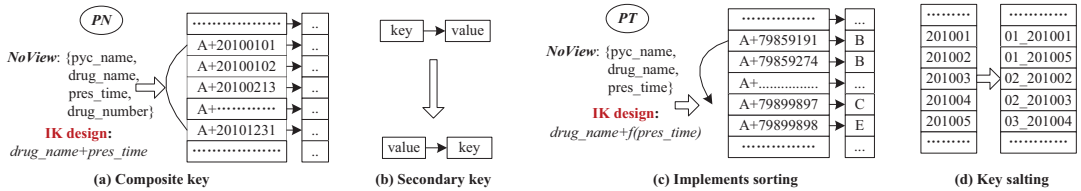
Secondary Key. For queries that need to access non-key data fields, it is straightforward to maintain a secondary key-value table. As illustrated in Fig. 7(b), the idea is to create and maintain another table (i.e., the value-key table) with secondary keys that follow the access pattern.

Implements Sorting. For queries with the *sort* operator, the trait of NoSQL's naturally sorted can be utilized. As shown in Fig. 7(c), the *PT* query needs to get the most recent data. The intelligent key reverses timestamp's order (with the function $f(dateTime)$ we mentioned before) to match HBase's ascending lexicographical order. This intelligent key design will create the property of being able to do a few scans to quickly obtain the most recently records.

Key Salting. NoSQL splits data among multi-servers. Data in the same split is lexicographically ordered to store related rows together. This design potentially leads to servers unavailability. For example, the time series data will lead to a large amount of traffic for one specific server. To avoid it, particular tags or random data will be added to the start of the key to write data into multiple data splits across the cluster, rather than one at a time, as demonstrated in Fig. 7(d). Formally, this intelligent key design pattern is called key salting.

Automatic *IK* design for *NoView*. According to the predicates in the query (i.e., the parameters in the *where* clause) on *NoView* and the indexes created in relational database, our query-aware approach can automate the *IK* design for *NoVIew*. In the running example, two columns (i.e., *drug_name* and *pres_time*) are indexed before migration and the sets of predicates for two queries are {*drug_name*} and {*drug_name*, *pres_time*} respectively. We briefly describe our data modeling algorithm as the following steps.

(1) *Migrating Indexes: one index to one secondary key.* If a parameter is in the set of query predicates and the corresponding column attribute is indexed in relational database before



(a) Composite key (b) Secondary key (c) Implements sorting (d) Key salting

Fig. 7. Common Patterns for Intelligent Key Design

migration, applying the secondary key pattern for the parameter to implement the index mechanism in NoSQL. Therefore, we can get one secondary key ($IK_{PT} = drug\_name$) for the *PT* query, and two ($IK1_{PN} = drug\_name$ and $IK2_{PN} = pres\_time$) for the *PN* query.

(2) *Parameter composition.* For the conjunctive parameters that are connected by the keyword *and*, applying the composite key pattern to combine all point parameters and one range parameter together (if there are more than one range parameters, our approach chooses one of them). It should be mentioned that the range parameter should be placed in the last order. That is, $IK_{PT} = drug\_name$ and $IK_{PN} = drug\_name + pres\_time$.

(3) *Combining the attribute with sort operator.* Applying the sorting key pattern to cope with the *sort* operators. Then, for the *PT* query, $IK_{PT} = drug\_name + f(pres\_time)$, in which f is the function that encodes the string, which is described in previous sections according to the descending order.

(4) *Redesigning range key for NoView reuse.* Different queries may share a same *NoView*, but their storage styles may differ. Without changing the result (i.e., the range scan from start key to end key is the same as that from end key to start key), the row key for range parameter can be reversed to help the *NoView* reuse. In the end, $IK_{PT} = IK_{PN} = drug\_name + f(pres\_time)$.

The algorithm focuses on the *IK* design for *NoView* (note: we omit the description on HBase's internal structure as it is trivial and has light effects on performance). The first step is general and the later steps figure out more compact form.

V. Evaluation

We carried out the migration project from June 2013 to May 2014, with nearly 11 months. The migration was from Oracle-11g on a single server (and a backup server) to a local cluster of 20 DELL OptiPlex-990 nodes. Each node is equipped with Intel i7-2600 3.4GHz cores, 16GB RAM and 12TB hard disk drives. The operating system in each node is Ubuntu-11.04 x86_64. The target Hadoop ecosystem includes Hadoop-1.2.1, HBase-0.92.0, Hive-0.9.0 and Sqoop-1.4.4.

Methodology. We do not make direct comparisons between the solutions of relational database and Hadoop ecosystem, because it is comparing apples to oranges. In practice, as it is hard to quantify the requirements on parametric queries' response times. Apart from the documents for quality assurance, we also interacted with application maintainers from our industry partner and interviewed 42 users to determine whether the query performance is "satisfactory".

TABLE II
Storage and Performance in Relational Database

Relational Database Solution			
Storage		Performance	
Basic Table	MV	Maintenance	Query
67 (11.4TB)	26 (4.8TB)	$> 5hr \uparrow$	121(37,84,27), 52.9%

Table II lists the basic information before migration. HEALTH contains 67 relational tables which occupy about 11.4TB data storage, and 26 materialized views with 4.8TB data size. It usually spent more than 5 hours ($\uparrow$ means the time keeps increasing) to maintain the data consistency. There are 121 queries, including 37 interactive queries and 84 long running queries. Among the interactive queries, there are 27 ones, of which the response time is closely related with users' input parameters. Before migration, the satisfactory rate for performance is $(28 + 39)/121 = 52.9\%$. It should be mentioned that, there are 24 reports that are most frequently produced. But the average cost is more than 6 hours.

This section makes the evaluation to answer three research questions. (1) What the migration brings and how does data model transformation benefit the migration? (2) How about the adoption of different techniques and design patterns in migration? (3) Are there any practical guidelines or lessons learned from the experience for developers?

A. Migration Effects

The migration to Hadoop ecosystem gains the overall advantage on scalability, high performance, and elasticity for future demands. Due to the designed scalable architecture of Hadoop ecosystem, the scalability can be inherently achieved after migration. For the service extension, we have started the machine learning tasks with Mahout[9], the framework based on Hadoop. Below we concentrate on two important aspects: the data storage and query performance.

Table III shows the results of two storage solutions for data migration in Hadoop ecosystem. (1) Data storage. It should be mentioned that Hadoop has 3 backups for data to get the fault tolerance, therefore the real data size is three times the size of the number listed in table. With the hybrid storage solution, there is about $(5.02-4.8)/4.8 = 4.6\%$ extra storage cost when generating *NoView* for interactive queries. (2) Performance.

[9]Apache Mahout. http://mahout.apache.org

TABLE III
STORAGE AND PERFORMANCE IN THE SOLUTION OF HADOOP ECOSYSTEM

HDFS				Hybrid: HDFS+HBase			
Storage		Performance		Storage		Performance	
Basic	Intermediate	Maintenance	Query	Basic	Intermediate	Maintenance	Query
11.3TB	26 (4.8TB)	$< 2hr$	65.3%	11.3TB	33 (5.02TB)	$< 2hr$	100%

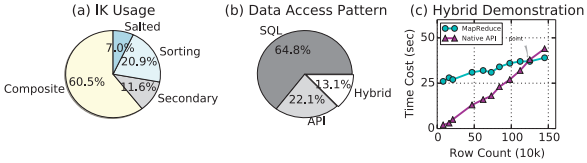


Fig. 8. Technique Adoption

The maintenance tasks have slight impact on applications, for they are performed in a batch way during midnight. Due to the computing scalability of MapReduce, the tasks can always be accomplished within certain time span ($< 2hr$) and the maintenance time keeps steady with incremental data size. After migration, the average cost for the 24 most frequent reports is reduced to about 80 minutes. To take the advantages of Hive (on HDFS) and HBase, the hybrid storage solution can gain the 100% satisfactory rate on query performance over the 65.3% with the sole HDFS.

Summary. Migrating to Hadoop ecosystem achieves the motivational effects. Though additional storage is required given the preference for fault tolerance and extra data beyond basic tables, data size is no longer considered to be the main bottleneck for most companies. Moreover, data model transformation benefits the migration with performance improvement according to requirements of different query types.

B. Adoption of Techniques and Patterns

NoView and *IK*. There are 27 *NoView*s generated for 37 interactive queries in HEALTH. Among them there are 5 ones can be reused for more than one queries. For intelligent key, there are 43 ones adopted totally. One query may employ more than one intelligent key patterns, such as the *PT* query. Fig. 8(a) demonstrates the distribution of different intelligent key patterns' usage. The first two intelligent key design patterns (i.e., the composite key and implementing sort) account for a large proportion (more than 80%). The salted key is specially used to avoid data skew when incrementally updating *NoView*.

Data access pattern after migration. When generating *NoView*, the initial SQL will be simplified by its pre-computation. Then after the *IK* phase, data access pattern will be either native HBase API or MapReduce (for range scan). Fig. 8(b) shows the proportions of them. The hybrid pattern entails fine-grained tuning, mainly for the queries of which the performance is closely related to the range of parameter. Fig. 8(c) shows the query response time with native API and MapReduce under different row counts. When the data is small (less than $1250k$ rows), the native API outperforms the batch-oriented MapReduce. Conversely, MapReduce is more appropriate under huge row counts for its parallelism. Our experiments confirm Phoenix's open testing results[10].

C. Lessons Learned

During our migration practice, we have learned the following impressive lessons.

(1) *Data structures and algorithms in relational database.* Hadoop ecosystem sacrifices most sophisticated mechanisms in relational database. In our practice to achieve the similar benefits, we realized the importance of data structures and algorithms behind these mechanisms. For example, the *NoView* is a special kind of materialized view and the *IK* implements the indexes. To some extent, the migration is also the process of implementing these techniques.

(2) *Major shift in data modeling methodology.* In our practice, we found that a number of developers were still accustomed to the principle for relational database, and followed the thinking in the domain of NoSQL. It brought inefficient results in most time. The principle for relational database is driven by structure of available data and the main theme is "design for answers", relying on rigid adherence to database schema, normalization and joins. However, solutions based on NoSQL are custom made. It is driven by application-specific data access patterns. The main methodology can be summarized as "design for questions". Data is duplicated and de-normalized as relationship-less [21].

(3) *Knowledge on management of Hadoop ecosystem.* The architecture based on Hadoop ecosystem brings much pain on system management. In 2014, we had coped with 11 non-technical issues, such as the failover of HBase's region servers, MapReduce job's out of memory error and so on, which motivate another line of our research [24], [25]. Due to the big gap in administration aspects (i.e., installation, monitoring and etc.) between the open-source Hadoop ecosystem and the mature relational database product, developer team's knowledge on Hadoop ecosystem is a key factor that should be considered before migrations. Meanwhile, it is important for enterprises to keep a team of maintainers.

VI. RELATED WORK

Hadoop ecosystem enhancement. There is a big gap in adminstration tasks (like installation, configuration, maintenance, monitoring) between mature relational database products and the open-source Hadoop ecosystem. In recent years, the gap motivates much effort on assisting approaches and auxiliary tools. For example, Shang *et al.* [13] proposed a testing-based

[10]Phoenix Performance Testing. http://phoenix.apache.org/performance.html

approach to uncover the different behavior of the underlying platforms for big data analytic applications between runs with small testing data and large real-life data for Hadoop. Another topic is the study [35], testing [31] and application [30], [41] of MapReduce programs. Specifically for NoSQL databases, the SOS platform [4] implements a common programming interface based on a meta-modeling method that maps the specific interfaces of the individual systems to a common one. Michael *et al*. [40] propose a cost-driven approach to optimize query performance while minimizing storage overhead. The core method is to use the cost of executing a given workload for a given schema to guide the data model design. Most recently, a web-based tool (KDM) [21], which advocates the query-driven methodology, is implemented for Cassandra[11] to visualize and support data modeling process.

Schema evolution. Database schema evolves as its application. Schema evolution is an extensively studied topic, yielding various techniques and tools [27], [28], [29]. In NoSQL world, the importance of schema evolution has also been recognized due to the sweet spot on data model's flexibility. [22] defines a declarative language for NoSQL schema evolution and supports common operations. ControVol [42] is implemented to integrate IDEs to detect the schema evolution related problems. Though schema evolution is not the focus in this paper, the strategies proposed provide potential methods for further data evolving in NoSQL database after migration.

Query optimization in NoSQL. Industrial solutions tend to develop generic index structures in server side, like HuaWei's hindex[12] and so on. In most cases, index is implemented in application side, which motivates considerable research work. For example, geographical queries (e.g., *KNN*) in location based service (LBS) applications apply dimension reduction in NoSQL [38]. Von *et al*. [33] adopt a heuristic strategy to build up the secondary index for keywords according the query distribution. Sfakianakis *et al*. [34] propose a hybrid approach to combine the segment tree based index and endpoint index for interval queries. Nikos *et al*. present a suite of solutions to optimize rank-join queries [36], varying from none index to composite row key based index.

SQL to NoSQL transformation. The problem of SQL-to-NoSQL data model transformation has raised widespread concerns, which appear mostly in developer forums, blog posts and presentations that focus on best practices, common use cases and sample designs [21]. Current SQL engines implement adapters to extend the table storage from default HDFS to NoSQL, such as Hive's *StorageHandler*, JackHare [23] and so on. However, the strategy is data-oriented with "one-to-one" mapping scheme and focuses on the execution layer with different computing paradigms. Rather than relying on the adapters, real-world migrations are usually conducted in a manual way with custom-developed programs. The migration experiences (e.g., Netflix[13] and [7]) share lessons and provide

[11]Apache Cassandra. http://cassandra.apache.org
[12]Hindex. Secondary Index. https://github.com/Huawei-Hadoop/hindex
[13]http://media.amazonwebservices.com/Netflix_Transition_to_a_Key_v3.pdf

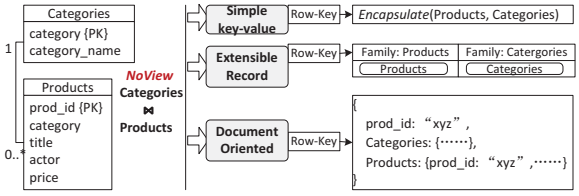


Fig. 9. Illustrative Example for Generality Study

guidelines such as "de-normalizing many-to-many relationship tables to flat-wide tables", "putting attributes together", and so on. Zhao *et al.* [8] propose a schema conversion model, which dedicates on high performance of the join operations by nesting relevant tables. They implement a graph transforming algorithm to contain all required content. Serrano *et al.* [26] develop a heuristic method for transforming relational schema to HBase with four steps: relation de-normalization, extended table merging, key encoding and views based on indexes. They also consider data access patterns to improve the transformation quality in a post-processing phase. These work conducts the data-first transformation without knowing any queries. The common way can be summarized as: to keep basic tables or to join all relationships (i.e., one-to-one, one-to-many and many-to-many) to get flat-wide tables for any upcoming ad-hoc queries. The so-called "nesting" techniques are indeed to join of attributes and encapsulate them into the nested internal structures in NoSQL's data model. Compared with them, our approach is specifically driven by the queries to get fine-grained results rather than the method of "universal" tables first and heuristic optimizations later. Moreover, our approach can also cover these work by defining a special *NoView* according to the above common way. From another perspective, our approach bridges both relational view system (Hive) and NoSQL system (HBase) in the same architecture, which is similar to that of CoSQL [39].

VII. Discussion

A. Generality

This section makes an in-depth study on the generality of our transformation approach. We borrow two relational tables from the DVD selling example used in [26], as shown in Fig. 9. From a methodology perspective, the two fundamental components (i.e., *NoView* and *IK*) are the abstractions of professional data modeling manners for expert NoSQL developers.

- *Insights in NoView. NoView* reflects the query-driven disciple to determine data elements stored in NoSQL, rather than the data-first manner and Normal Form theory for relational database. In different situations, *NoView* can be obtained or calculated from different inputs or conceptual models. For instance, our approach extracts *NoView* through the well-expressed SQL language as a reverse engineering process from the traditional solution. Another typical scenario is the design phase for a new application based on requirement specifications, queries may be documented as natural

language (e.g., "*Find products and category information of the product with ID 2179*" in [26]). In such case, *NoView* (i.e., underline attributes) need to be figured out by other corresponding techniques and tools.

- *Insights in IK.* Previous studies [7], [21], [33], [26] have pointed out the importance of row-key to decide data structures in NoSQL. Given a query, *IK* attempts to take best benefits of data entrance to implement as more operators in the storage layer as possible. Otherwise, row-key turns into a unique identifier for a record. From another view, row-key can be seen as the connector to join different attributes in NoSQL's internal data structures, such as different column families indexed by the same row-key in HBase.

Though our approach is proposed against the background of data migration, the essence of two fundamental components decides its generality in other application areas. Even the components may not be combined as a full automatic approach, they can be extended or separately adopted. Below we discuss how to generalize our approach to the other cases.

Generalize to the case with no query. Despite of NoSQL's query-driven nature, there are many application areas that require to design data model first without knowing any queries. In such case, there are two frequent ways: (i) keep basic tables or (ii) join all relationships (i.e., one-to-one, one-to-many and many-to-many) to get flat-wide tables for any upcoming ad-hoc queries. Therefore, we can extend *NoView* to cover this case by defining a special view under no query input. For example, *NoView* is the join of two tables in Fig. 9. As there is no query, the row-key plays the role as a unique identifier, without applying *IK* design techniques.

Generalize to other NoSQL systems. There are many kinds of NoSQL systems. With respect to data model, a very common way is to classify them into (1) simple key-value stores, (2) documents stores and (3) extensible records stores [22], [33]. In a unified way, we can denote them as *key-structure*. The *structure* part represents different data structures accordingly to different NoSQL databases, for example the nested *column family: column* in HBase and *json* in document-oriented MongoDB[14]. The function of the *structure* part is to organize logic-related attributes together, which is equivalent to the *join* operation in query layer. In our approach, *NoView* is independent with concrete NoSQL data model, and *IK* concentrates on the common row-key part for all NoSQL data model variants. Though we put forward the techniques and patterns based on HBase, they can be easily generalized to other NoSQL systems. Fig. 9 shows the actions in different NoSQL databases to implement the logic in *NoView*.

Generalize to other application types. The query-aware approach tailored to the context of data analytics applications, and functions on read-only query workloads. For other applications that contain updates which implicitly constrain the amount of denormalization, a heuristic strategy on tradeoff of maintenance cost and query performance is necessary. In such cases, though the approach cannot be applied directly, by incorporating a cost model [40], the method of *NoView* and *IK* design patterns are still appropriate. For extension purpose, *NoView* can be seen as a "core" for further schema evolution operations (e.g., to move or add attributes based on *NoView*).

[14]https://www.mongodb.org

B. Threats to validity

Threats to internal validity concern our selection of big data frameworks, classification of workload types and metrics used to evaluate migration quality. There are many other available frameworks in Hadoop ecosystem and our approach does not consider platform-specific techniques, such as the CQL language for Cassandra, leading to the loss of some unique optimizations chances. The metrics we used to evaluate the transformation quality are data storage and query performance in the context of data analytics applications. However, they may be different for other kinds of applications. For example, data equivalence [26] is also an important metric for applications that involve data modification.

Threats to external validity concern the possibility to generalize our work and results. To begin with, it is not safe to say that Hadoop ecosystem is suitable for all big data applications. The work on data analytics application in this paper may not provide enough diversity in the applications to ensure generality of our conclusions, for example to other application types. To address this threat, more case studies in the application migrations have to be conducted.

VIII. Conclusion and Future Work

Migrating legacy applications to more modern platforms is a recurring software development activity. In the big data era, more than ever enterprises are confronted with the data-driven software maintenance and evolution challenges. This paper presents the migration of a real-world data analytics application to Hadoop ecosystem. We present the architecture re-design to demonstrate the method in big data environment. We focus on the SQL-to-NoSQL data model transformation problem and propose an automatic query-aware approach to free developers from tedious manual work. We believe that our work can provide insightful guidelines for other migrations.

Our future work can be divided into two aspects. The first one is to develop a unified query engine, which inherently supports multiple heterogeneous data storages. The other one is the further study on SQL-to-NoSQL data model transformation problem. With the widely adoption of NoSQL databases, this topic is worthy of attention. Here, we raise some research problems. (1) Domain specific languages (DSL) and tools with heuristic rules to support transformation. (2) Automatic approaches with cost model for other specific applications.

Acknowledgment

We acknowledge the anonymous reviewers for their insightful comments and suggestions. This work was supported by Chinese Academy of Sciences STS Project (KFJ-SW-STS-155), Major Programs of the General Logistics Department (AWS14R013), and National Key Research and Development Plan Program (2016YFB1000103).

REFERENCES

[1] H. M. Chen, R. Kazman, S. Haziyev and O. Hrytsay, "Big Data System Development: An Embedded Case Study with a Global Outsourcing Firm," in *Proceedings of the 1st International Workshop on Big Data Software Engineering (BIGDSE/ICSE)*, 2015, pp. 44-50.

[2] S. Ghemawat, H. Gobioff and S. T. Leung, "The Google File System," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 29-43.

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," in *6th Symposium on Operating System Design and Implementation (OSDI)*, 2004, pp. 137-150.

[4] P. Atzeni, F. Bugiotti and L. Rossi, "Uniform Access to Non-relational Database Systems: The SOS Platform," in *Proceedings of the 24th International Conference on Advanced Information Systems Engineering (CAiSE)*, 2012, pp. 160-174.

[5] K. Harezlak and R. Skowron, "Performance Aspects of Migrating a Web Application from a Relational to a NoSQL Database," in *Proceedings of the 11th International Conference Beyond Databases, Architectures and Structures (BDAS)*, 2015, pp. 107-115.

[6] Y. Wang, Y. Z. Xu, Y. Liu, J. Chen and S. L. Hu, "QMapper for Smart Grid: Migrating SQL-based Application to Hive," in *Proceedings of the Conference on Management of Data (SIGMOD)*, 2015, pp. 647-658.

[7] A. Schram and K. M. Anderson, "MySQL to NoSQL: Data Modeling Challenges in Supporting Scalability," in *Proceedings of the Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, 2012, pp. 191-202.

[8] G. S. Zhao, L. B. Li, Z. J. Li and Q. Y. Lin, "Multiple Nested Schema of HBase for Migration from SQL," in *Proceedings of the Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, 2014, pp. 338-343.

[9] M. Stonebraker and U. cetintemel, "One Size Fits All: An Idea Whose Time Has Come and Gone," in *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 2011, pp. 2-11.

[10] H. Herodotou, F. Dong and S. Babu, "No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics," in *Proceedings of the Symposium on Cloud Computing (SOCC)*, 2011, pp. 18-18.

[11] C. Bondiombouy, B. Kolev, O. Levchenko and P. Valduriez, "Integrating Big Data and Relational Data with a Functional SQL-like Query Language," in *Proceedings of the 26th International Conference on Database and Expert Systems Applications (DEXA)*, 2015, pp. 170-185.

[12] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis and M. J. Carey, "MISO: Souping up Big Data Query Processing with a Multistore System," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2014, pp. 1591-1602.

[13] W. Y. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan and P. Martin, "Assisting Developers of Big Data Analytics Applications when Deploying on Hadoop Clouds,", in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 402-411.

[14] S. Agrawal, S. Chaudhuri and V. R. Narasayya, "Automated Selection of Materialized Views and Indexes in SQL Databases," in *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, 2000, pp. 496-505.

[15] R. DeLine, "Research Opportunities for the Big Data Era of Software Engineering," in *Proceedings of the 1st International Workshop on Big Data Software Engineering (BIGDSE/ICSE)*, 2015, pp. 26-29.

[16] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, F. Chakka, N. Zhang, S. Anthony, H. Liu and R. Murthy, "Hive-a Petabyte Scale Data Warehouse Using Hadoop," in *Proceedings of the 26th International Conference on Data Engineering (ICDE)*, 2010, pp. 996-1005.

[17] F. Chang, J. Dean, S. Ghemawat, D. Wallach, M. Burrows, T. Chandra, A. Fikes and R. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 205-218.

[18] F. Matthes, C. Schulz and K. Haller, "Testing & Quality Assurance in Data Migration Projects," in *Proceedings of the 27th International Conference on Software Maintenance (ICSM)*, 2011, pp. 438-447.

[19] H. Z. Yang and P. A. Larson, "Query Transformation for PSJ-Queries," in *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB)*, 1987, pp. 245-254.

[20] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh and M. Urata, "Answering Complex SQL Queries Using Automatic Summary Tables," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2000, pp. 105-116.

[21] A. Chebotko, A. Kashlev and S. Y. Lu, "A Big Data Modeling Methodology for Apache Cassandra," in *Proceedings of the IEEE Congress on Big Data (BigDataCongress)*, 2015, pp. 238-245.

[22] S. Scherzinger, M. Klettke and U. Storl, "Managing Schema Evolution in NoSQL Data Stores," in *Proceedings of the 14th International Symposium on Database Programming Languages (DBPL)*, 2013.

[23] W. C. Chung, H. P. Lin, S. C. Chen, M. F. Jiang and Y. C. Chung, "JackHare: a Framework for SQL to NoSQL Translation Using MapReduce," *Automated Software Engineering (ASE)*, 2014, 21(4):489-508.

[24] L. Xu, J. Liu and J. Wei, "FMEM: A Fine-grained Memory Estimator for MapReduce Jobs," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC)*, 2013, pp. 65-68.

[25] L. Xu, W. Dou, F. Zhu, C. Gao, J. Liu, H. Zhong and J. Wei, "A Characteristic Study on Out of Memory Errors in Distributed Data-Parallel Applications," in *Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 518-529.

[26] D. Serrano, D. Han and E. Stroulia, "From Relations to Multi-dimensional Maps: Towards an SQL-to-HBase Transformation Methodology," in *Proceedings of the 8th International Conference on Cloud Computing (Cloud)*, 2015, 81-89.

[27] G. Papastefanatos, F. Anagnostou, Y. Vassiliou and P. Vassiliadis, "Hecataeus: A What-if Analysis Tool for Database Schema Evolution," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*, 2008, pp. 326-328.

[28] A. Cleve, J. Henrard, and J. L. Hainaut, "Data Reverse Engineering using System Dependency Graphs," in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE)*, 2006, pp. 157-166.

[29] L. Meurice and A. Cleve, "DAHLIA: A Visual Analyzer of Database Schema Evolution," in *Proceedings of the Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 464-468.

[30] W. Y. Shang, B. Adams and A. E. Hassan, "An Experience Report on Scaling Tools for Mining Software Repositories Using MapReduce," in *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, 2010, pp. 275-284.

[31] C. Csallner, L. Fegaras and C. K. Li, "Testing Mapreduce-style Programs," in *Proceedings of the 19th SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2011, pp. 503-507.

[32] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem and P. Helland, "The End of an Architectural Era (It's Time for a Complete Rewrite)," in *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, 2007, pp. 1150-1160.

[33] C. von der Weth and A. Datta, "Multiterm Keyword Search in NoSQL Systems," *IEEE Internet Computing*, 2012, 16(1):34-42.

[34] G. Sfakianakis, I. Patlakas, N. Ntarmos and P. Triantafillou, "Interval Indexing and Querying on Key-Value Cloud Stores," in *Proceedings of the 29th International Conference on Data Engineering (ICDE)*, 2013, pp. 805-816.

[35] T. Xiao, J. X Zhang, H. C Zhou, Z. Y. Guo, S. McDirmid, W. Lin, W. G Chen and L. D. Zhou, "Nondeterminism in MapReduce Considered Harmful? An Empirical Study on Non-commutative Aggregators in MapReduce Programs," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 44-53.

[36] N. Ntarmos, I. Patlakas and P. Triantafillou, "Rank Join Queries in NoSQL Databases," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2014, pp. 493-504.

[37] F. Villanustre, "Industrial Big Data Analytics: Lessons from the Trenches," in *Proceedings of the 1st International Workshop on Big Data Software Engineering (BIGDSE/ICSE)*, 2015, pp. 1-3.

[38] N. Dimiduk, A. Khurana, M. H. Ryan and M. Stack. HBase in Action. Shelter Island: Manning, 2013.

[39] E. Meijer and G. M. Bierman, "A Co-relational Model of Data for Large Shared Data Banks," *Commun. ACM*, 2011, 54(4):49-58.

[40] M. J. Mior, "Automated Schema Design for NoSQL Databases," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2014, pp. 41-45.

[41] J. J. Stephen, S. Savvides, R. Seidel and P. Eugster, "Program Analysis for Secure Big Data Processing," in *Proceedings of International Conference on Automated Software Engineering (ASE)*, 2014, pp. 277-288.

[42] S. Scherzinger, T. Cerqueus and E. C. Almeida, "ControVol: A Framework for Controlled Schema Evolution in NoSQL Application Development," in *Proceedings of the 31th International Conference on Data Engineering (ICDE)*, 2015, pp. 1464-1467.