Transforming Relational Database into HBase: A Case Study

Chongxin Li
College of Computer Science and Technology
Zhejiang University, Hangzhou, China
lichongxin@zju.edu.cn

Abstract— **With the development of distributed system and cloud computing, more and more applications might be migrated to the cloud to exploit its computing power and scalability, where the first task is data migration. In this paper, we propose a novel approach that transforms a relational database into HBase, which is an open-source distributed database similar to BigTable. Our method is comprised of two phases. In the first phase, relational schema is transformed into HBase schema based on the data model of HBase. We present three guidelines in this phase, which could be further utilized to develop an HBase application. In the second phase, relationships between two schemas are expressed as a set of nested schema mappings, which would be employed to create a set of queries or programs that transform the source relational data into the target representation automatically.**

Keywords-HBase; data transformation; schema mapping;

I. INTRODUCTION

Cloud has become a trend for both data storage and processing because of its agility and scalability. By using PaaS (Platform-as-a-Service), companies especially the small ones, just pay for what they use rather than have to maintain the underlying platform. Furthermore, applications can scale up and down on an as-needed basis without extra efforts. These benefits make cloud a good choice for applications whose throughput varies a lot. However most cloud platforms don't support relational data for the consideration of scalability. To meet the reliability and scaling needs, several storage technologies have been developed to replace relational database for structured and semi-structured data, e.g. BigTable [3] for Google App Engine (GAE) and SimpleDB[10] for Amazon Web Service (AWS). As a result, relational database of existing applications must be transformed into these cloud-base databases so that they can run on such platforms.

Data transformation and integration is an old research topic which obtains more and more interests in recent years because of the increased need of data exchange in various formats. It is initially developed to cope with relational schemas [6, 9] and then extended to deal with nested XML data [1, 4]. Further research then mainly focus on the improving of semantics consistency [2]. At the same time, prototype system is developed based on these researches to automate the data transformation. For example Clio [7, 9], which is partly integrated into IBM's DB2, can generate mappings between any combinations of relational and XML schemas. However, to the best of our knowledge, little work has done to transform a relational database into a non-relational cloud-based database, which is main scenario of this paper.

We present a novel solution as well as a case study in this paper for transforming relational database into cloud database. Comparing to traditional normalized database which follows the *Normal Forms*, almost all cloud databases are denormalized: data might be grouped or replicated to accelerate the read speed. Although database becomes consistent eventually, there might be inconsistent data during a short time for an operation. Therefore, cloud databases are most suitable for read-heavy OLAP systems where short data inconsistency is tolerable, such as web applications. On the other side, OLTP system where ACID is very important should not be moved to the cloud, e.g. banking system. We start by examining one of the cloud-based databases——HBase, which is an open-source implementation of BigTable [5]. Based on the data model and features of HBase, we present a heuristic transformation solution. Although our work is only applicable to HBase, BigTable and similar cloud-based databases (Hypertable, Cassandra), this paper still might be seen as a step in this direction and a hint for the future work.

The key contributions of our work are summarized as follows:

- We identify the problem of data transformation between relational database and cloud-based database
- We propose a heuristic-based solution for transforming relational database to HBase, which is one of the cloud databases.

The rest of this paper is organized as follows. Section II briefly surveys the HBase data model. In Section III, we present our transformation solution using a case study. Finally, Section IV concludes the paper with some directions for the future work.

II. PRELIMINARY

A BigTable is a sparse, distributed, multidimensional sorted map [3]. As an open-source implementation of BigTable, HBase uses a data model very similar to that of BigTable. A data row in HBase has a sortable *row key* and an arbitrary number of columns, which are further grouped into sets called *column families*. Each data cell in HBase can contain multiple versions of the same data which are indexed by timestamp. If not specified, data with the latest timestamp will be read and written by default. Thus in the rest sections of this paper, we'll not consider timestamp explicitly. To differentiate from

978-1-4244-6055-7/10/$26.00 ©2010 IEEE

traditional relational databases whose data is usually represented as a table form, we represent an HBase table here as a sorted map (Fig. 1(a)). Consequently, a data cell in the table can be viewed as a key value pair where the key is a combination of row key, column and timestamp; and the value is an uninterpreted array of bytes (Fig. 1(b)).

```
RowKey{
   Column Family A{
      Column M:
         t1   Value1
         t2   Value2
      Column N:
         t3   Value3
   }
   Column Family B{
      Column X:
         t4   Value4
         t5   Value5
         t6   Value6
   }
}
```

(a)

(Row, Family:Column, Timestamp) →Value

(b)

Figure 1. HBase data model

Column family is a concept introduced by BigTable and also plays an important role in HBase. Data saved in the same column family are stored together on the file system while data for different column families might be distributed on several machines. For this reason, HBase is also referred as a column-oriented database. A column family must be created before data can be stored in it. And also it is assumed that the number of column families is small (in the hundreds at most) and families rarely change during operation. On the contrary, a table may have unbounded number of columns and a column can be added into a family dynamically. This makes both column key and value applicable to store user data. Therefore different rows of a table must have the same column families but might have completely different columns. HBase data model and all the above features should be taken into consideration for the later transformation or when you design an HBase table.

III. TRANSFORMING RELATIONAL DATABASE INTO HBASE

A. Overview

To better understand the transformation of relational database, we start the problem with an example.

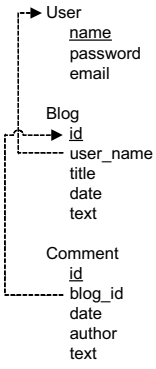

Figure 2. Example Schema: Blog System

Example Consider the above schema in Fig. 2. It represents a blog system with three tables: *User* (name, password, email), *Blog* (id, user_name, title, date, text) and *Comment* (id, blog_id, date, author, text). As with the many blog systems, each user can post a blog on the system, and anyone even those who are not the users of this system can post a comment to a blog. Foreign keys are shown as dash lines. Then our problem is how to transform such a relational database into HBase without information loss.

We divide the transformation process into two phases. In the first phase, a heuristic approach is proposed to transform relational schema of the source database into HBase schema based on the data model and features of HBase. Then the second phase generates the mappings between the source schema and target schema. Since values are grouped in column families, we extend the technique of nested mapping [4] which has shown pretty good result in XML schema mapping. Thus data from the source database can be transformed into the target representation based on the schema mappings generated in the second phase. The whole process is demonstrated as Fig. 3. These two phases are described in details in the following two sections. It is worth noting that, although we illustrate our approach using a case study, our method is also applicable to other relational databases.

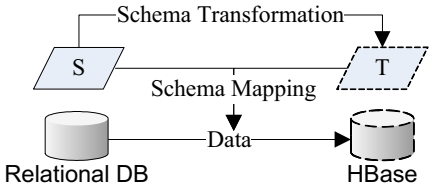

Figure 3. Transformation Overview

B. Schema Transformation

As described in the first section, data grouped in the same column family are saved together in the file system. Intuitively, this leads to the first rule for HBase schema design.

Rule 1: Group correlated data in a column family.

Take the blog system as an example. Usually a blog page is started with blog title and post time, and then content of the blog takes up the main part of the page. So all these information tend to be read and written together. For the same reason, if you view a user's information on his personal page, profiles about this user will be read together from the database. Therefore, this kind of information can be thought as *correlated*. Although relational schema is designed under normalization rule, it still gives some inspiration about the data relation. Thus a simple HBase schema can be derived directly from the relational schema; each original table in relational database corresponds to an HBase table where old primary keys are treated as row keys and only one column family is created. Columns of original table are grouped into the unique column family to guarantee that these related data are stored together. Further more we may have seen pages where blogs of the same user is listed in one page, only with titles, post time and a short paragraph probably. If you want to read the blog, you have to click the title and then content of the blog is

retrieved from the database. From this point of view, the "correlated data" mentioned in this rule is more about the access pattern and write pattern of the application, instead of the literalistic which is easy to understand. Then under such circumstance blog content and other meta information are not accessed together and can be split into two column families. The initial HBase schema for the blog system is shown as Fig. 4(a).

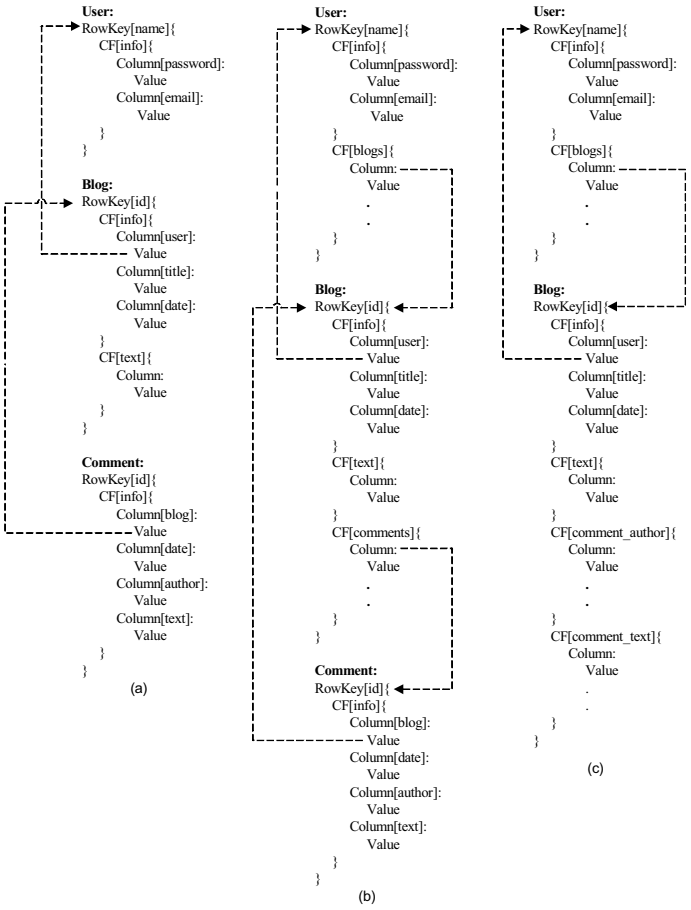

Figure 4. Schema Transformation after (a) ***Rule 1*** (b) ***Rule 2*** (c) ***Rule 3***

However, this initial schema is far from a good design of HBase schema. In relational database, there are three kinds of relationships: one-to-one, one-to-many (many-to-one), and many-to-many. Foreign keys are used to maintain such a relation and to reference parent and child objects. But in HBase, we can only use row keys to get data. Thus, it is difficult for an object to access other objects without knowing their row keys, e.g. in Fig. 4(a), we can simply get the author of a blog by the *user_name* column, but it is not that easy to retrieve all the blogs belonging to a specified user because the user doesn't have references to all his blogs and no *Join* operation is supported in HBase for the consideration of scalability. This leads to the second rule of HBase schema design.

Rule 2: For each side of a relationship, add the foreign key references of the other side if it needs to access the other side's objects.

We consider this rule for three kinds of relationships:

1) One-to-One relationship

Actually we don't have to worry about this kind of relationship in HBase because this has already been done in relational database, where foreign key can be put on both sides of the relationship. Such a foreign key column is treated as an ordinary column in HBase and can be grouped with other columns based on *Rule 1*.

2) One-to-Many relationship

In relational database, foreign keys can be only put on the *many* side of a one-to-many relationship since multiple values are not allowed in RDBMS because of *Normal Form 1*. But in HBase, multiple values can be grouped together in a column family. To reference objects on the *many* side, a new column family should be created on the *one* side to contain a set of foreign keys of the *many* side. Take Fig. 4(b) as an example, *blogs* family of table *User* contains all the blog ids which are posted by a user. Then getting all blogs of a user can be divided into two steps. In the first step, get the blog ids by the row key—user *name*; and second step further fetches all these blogs by id. The same method applies for the comments of a blog.

3) Many-to-Many relationship

Things are a little different for many-to-many relationship because such a relationship is maintained by a third table, where foreign keys for both tables are kept. However, since both sides of the relationship are the *many* side, we can follow what we did for the *many* side of one-to-many relationship. New column families are created in both tables to capture row keys of the other side. At the same time, the third table which is used to maintain the relationship can be removed, because its information has been expressed on both side of the relationship.

Although we still call these references as foreign keys, they are different from those of relational database. For relational database, it is guaranteed by the database itself that data is always in a consistent state and user data can not violate the foreign key constraint. For example, a blog can not be inserted with a user name which does not exist yet. But in HBase, applications have to maintain these references instead. When a blog is deleted from the system, first delete it from table *Blog* and then from the *blogs* column family of table *User*. Otherwise a lazy delete can be utilized, that is blog entry in the table *User* is not deleted until next read. For both methods, there is always a period when the database is inconsistent. This can be seen as a tradeoff for scalability.

On the other side, if an object doesn't access other objects in the application, we don't have to add the foreign keys for this object, e.g. if the blog system doesn't have the requirement that list all the blogs of a given user, we can omit the column family *blogs* which contains the foreign keys.

By now, operations related to foreign key reference can be finished in two steps. Relational database design always breaks data into several tables to meet the *Normal Forms* and reduce data replication. That's one of the reasons why we have foreign keys. However, database is no longer normalized in cloud. These broken up data can be merged together to reduce foreign key references, and also reduce the two step operations to one. This is the third rule of our transformation.

Rule 3: Merge attached data tables to reduce foreign keys.

Usually, given a relational database of an application, there are tables whose data are more important and can be used independently; and tables whose data are accessory and must depend on other tables based on the foreign key. If a table can be used independently in the application, we call this table a *main table*. On the other side, if a table has only one foreign key and data of it must be used depending on the referenced object, such a table is an *attached table*. Data with the same foreign key in the attached table can be merged into a single row of the main table based on the foreign key. Let's have a look at the blog system again. To merge tables together, we must decide which one is the main table and which one is the attached table. Comments of a blog must be displayed with the content. If a blog is deleted, so are all its comments. Hence, *Blog* table is a main table and *Comment* table is an attached table accordingly. Next consider how the *Comment* table is merged into *Blog* table. A first thought is that a new column family *comments_full* is created in *Blog* table. Since columns in HBase can be added into family dynamically, each column in this family contains a row from the *Comment* table. To make data more understandable, we can use *date* of the comment as the column key and other fields such as author and text are combined into the column value. The weak point of this method is that the column value has to be parsed to display a comment. When these fields are of different types and the field length varies, it will be complicated to parse this semi-structured data. For simplicity we split the value fields, each with a new column family. The design is shown as Fig. 4(c). Although this might physically separate values of comment, comments are well structured and more convenient to access. This tradeoff is worthy.

We give some basic guidelines on how to transform a source relational schema into a target HBase schema. However, the transformation of relational schema should depend on the business of the particular application. Moreover, access patterns as well as write patterns need to be carefully considered so that the design of HBase schema can take good care of the most frequent and most important uses.

C. Schema Mapping

Given the source schema and target schema which is generated in the above section, schema mappings express the relations between these two schemas. Schema mapping has been intensively studied for both relational and XML schema. Although data inside HBase is organized in tables, we can see from the data model that HBase schema is more like that of XML, which is nested. Thus an approach based on nested mapping is used in this section.

To perform transformation, we must understand how two schemas correspond to each other. There are numerous proposals for representing the inter-schema correspondences, in which element correspondence is the most widely used. Intuitively, an element correspondence is a pair of a source element and a target element which have the same value. Usually, a technique named schema matching is utilized to extract the correspondences. However, since the target schema is generated by schema transformation, the correspondences can be captured during this process. The correspondences are shown as Fig. 5.

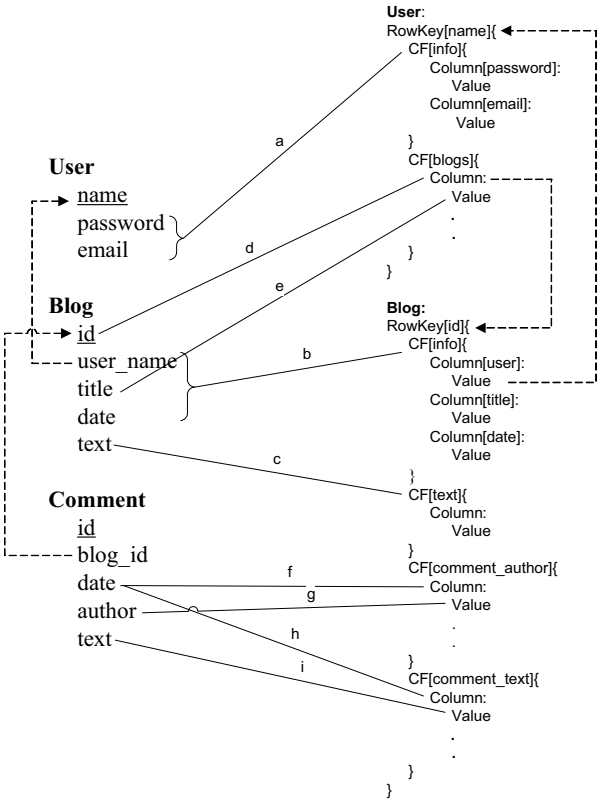

Figure 5. Schema correspondences between source schema (left) and target schema (right)

Next consider the generation of schema mappings. *Tableau* is a way of describing all the basic concepts and relationships that exist in a schema [4]. There is a set of tableaux for both source schema and target schema. Each tableau includes all related concepts, that is, concepts that must exist together according to the referential constraints. For the relational source schema, the tableau is computed the same way as before, by enhancing via the chase [8] with constraints. Source tableaux for blog system are shown in Fig. 6(a).

But things are a little different for the target HBase schema. Since columns can be added dynamically into the column family, columns are treated differently in two types. For the first type, column keys are treated as table meta data. This is the same as those of relational database and user data can be only put in column value. Correspondences *a, b, c* in Fig. 5 are of this type, such as data info:email = "tommy@gmail.com" in table *User*, column name *email* is fixed. However for the second type, both column key and value are used to contain user data, such as blogs:"hbase data model" = "24" in table *User*, where both "hbase data model" and 24 are user data. New column key and value are inserted into this family once a new blog is posted by this user. Thus this type of columns must be nested when generate the mappings, e.g. correspondences {*d, e*}, {*f, g*}, {*h, i*} in Fig. 5. Tableaux for the target schema are demonstrated in Fig. 6(b).

A_1 = { u[name, password, email] IN *User;* }
A_2 = { u[name, password, email] IN *User*, b[id, user_name, title, date, text] IN *Blog*;
b.user_name = u.name; }
A_3 = { u[name, password, email] IN *User*, b[id, user_name, title, date, text] IN *Blog*,
c[id, blog_id, date, author, text] IN *Comments*; b.user_name = u.name,
c.blog_id = b.id; }

(a)

B_1 = { u[rowkey, password, email] IN *User*; }
B_2 = { u[rowkey, password, email] IN *User*, b[colum, value] IN u.*blogs*,
b`[rowkey, user, title, date, text] IN *Blog*; b.user = u.rowkey, b.column = b`.rowkey; }
B_3 = { u[rowkey, password, email] IN *User*, b[rowkey, user, title, date, text] IN *Blog*;
b.user = u.rowkey; }
B_4 = { u[rowkey, password, email] IN *User*, b[rowkey, user, title, date, text] IN *Blog*,
ca[column, value] IN b.*comment_author*, ct[column, value] IN b.*comment_text*;
b.user = u.rowkey;}

(b)

Figure 6. Tableaux for Source Schema and Target Schema respectively

Then for each pair of tableaux (*A, B*), where *A* is a source tableau and *B* is a target tableau, let *V* be the set of all correspondences for which the source element is covered by *A* and the target element is covered by *B*. Triple (*A, B, V*) encode a basic mapping: for all the data covered by *A*, there exist data covered by *B* along with conditions that encode the correspondences. We may write the mapping (*A, B, V*) as $\forall A \rightarrow \exists B.V$. For our example $\forall A_2 \rightarrow \exists B_3.\{a,b,c\}$.

We now review the generation of nested mapping. The main concept is subtableau. A tableau *A* is a subtableau of a tableau *A'* if the data covered in *A* form a superset of the data covered by *A'* and also the conditions in *A* are a superset of those in *A'*. Then a basic mapping $\forall A_2 \rightarrow \exists B_2.V$ is nestable inside another basic mapping $\forall A_1 \rightarrow \exists B_1.V$ if the following hold:

1) A_2 and B_2 are strict subtableaux of A_1 and B_1 respectively.
2) V_2 is a strict superset of V_1.
3) There is no correspondence *v* in $V_2 - V_1$ whose target element is covered by B_1.

The result of nesting m_2 inside m_1 is a nested mapping of the form:

$$\forall A_1 \rightarrow \exists B_1.[V_1 \wedge \forall (A_2 - A_1) \rightarrow \exists (B_2 - B_1).(V_2 - V_1)] \quad (1)$$

For our example, we obtain the mappings:

M_1: $\forall A_1 \rightarrow \exists B_1.[\{a\} \wedge \forall (A_2 - A_1) \rightarrow \exists (B_2 - B_1).\{b,c,d,e\}]$
M_2: $\forall A_2 \rightarrow \exists B_3.[\{a,b,c\} \wedge \forall (A_3 - A_2) \rightarrow \exists (B_4 - B_3).\{f,g,h,i\}]$

Finally, we show below the nested mappings in query-like notations, which can be used to transform the data.

M1: user = *for* u *IN User*
return [
rowkey = u.name
info:password = u.password
info.email = u.email
blogs = *for* b *IN Blog*
where b.user_name = u.name
return [
column = b.id
value = b.title]]

M2: blog = *for* b *IN Blog*
return [
rowkey = b.id
info:user = b.user_name
info:title = b.title;
info:date = b.date;
text: = b.text
comment_author = *for* c *IN Comment*
where c.blog_id = b.id
return [
column = c.date
value = c.author]
comment_text = *for* c *IN Comment*
where c.blog_id = b.id
return [
column = c.date;
value = c.text]]

IV. CONCLUSION

In this paper, we have presented a heuristic-based approach for transforming a relational database into HBase. A case study is employed to demonstrate our proposed solution. For the first phase of our method, we propose three rules on how to transform a relational schema into an HBase schema. However, what really matter are the business requirements and access/write pattern of the applications. Currently, this work has to be done by application experts, but in the future we would provide some semi-automatic tool to simplify this process. For the second phase, we extend the nested mapping technique to make it adapted for HBase. There have been prototype systems for generating schema mappings between relational and XML schemas. It will not be too hard to integrate this extension into existing systems.

REFERENCES

[1] M. Arenas and L. Libkin, "XML data exchange: consistency and query answering," in Proceedings of the 28th ACM Symposium on Principles of Database Systems (PODS), 2005, pp. 13-24.

[2] L. Cabibbo, "On keys, foreign keys and nullable attributes in relational mapping systems," in Proceedings of the 12th International Conference on Extending Database Technology (EDBT), 2009, pp. 263-274.

[3] F. Chang., J, Dean., S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T, Chandra, A. Fikes and R. Gruber, "Bigtable: a distributed storage system for structured data," In Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2006, vol. 7, pp. 15-15.

[4] A. Fuxman, M. Hernandez, C. Ho, R. Miller, P.Papotti and L. Popa, "Nested mappings: schema mapping reloaded," in Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB), 2006, pp. 67-78.

[5] HBase. Web Page. hadoop.apache.org/hbase/

[6] P. Kolaitis, "Schema mappings, data exchange and metadata management," in Proceedings of the 24th ACM Symposium on Principles of Database Systems (PODS), 2005, pp. 61-75.

[7] R. Miller, M.Hernandez, L.Haas, L. Yan, C. Ho, R. Fagin, L. Popa, "The Clio project: managing heterogeneity," in ACM SIGMOD Record, 2001, vol. 30, pp. 78-83.

[8] L. Popa and V. Tannen, "An equational chase for path-conjunctive queries, constraints and views," in Proceedings of the 7th International Conference on Database Theory (ICDT), 1999, pp. 39-57.

[9] L. Popa, Y. Velegrakis, R. Miller, M.Hernandez, R. Fagin, "Translating web data," in Proceedings of the 28th International Conference on Very Large Data Bases (VLDB), 2002, pp. 598-609.

[10] E. Sciore, "SimpleDB: a simple java-based multiuser syst for teaching database internals," in Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education, 2007, vol. 39, pp. 561-565.