


RESEARCH

Open Access



Toward building RDB to HBase conversion rules

R. Ouanouki* , A. April, A. Abran, A. Gomez and J. M. Desharnais

*Correspondence:
Rafik.ouanouki.1@ens.etsmtl.ca
Software Engineering
and Information Technology
Department, ÉTS University,
Montreal, QC H3C 1K3,
Canada

Abstract

Cloud data stores that can handle very large amounts of data, such as Apache HBase, have accelerated the use of non-relational databases (coined as NoSQL databases) as a way of addressing RDB database limitations with regards to scalability and performance of existing systems. Converting existing systems and their databases to this technology is not trivial, given that RDB practitioners tend to use the relational model design mindset when converting existing databases. This can result in inefficient NoSQL design, leading to suboptimal query speed or inefficient database schema. This paper reports on a two-phase experiment: (1) a conversion from RDB to HBase database, a NoSQL type of database, without the use of conversion rules and based on a heuristic approach and (2) an experiment with different schema designs to uncover and extract conversion rules that could be useful conversion guidelines for the industry.

Keywords: Cloud computing, RDB to NoSQL conversion, Conversion rules, Conversion designs, Distributed databases, Database conversion

Background

There are different approaches proposed for addressing conversion between relational and NoSQL databases and most share the same goal: how to efficiently convert an existing relational database to a NoSQL database.

Lee et al. [1] proposed an automatic Sql-to-NoSQL schema transformation using MySQL and HBase as an example. The main objective of the proposed solution was to avoid cross-table queries. In this proposal, all tables that have a relationship were transformed into a single HBase table. This approach breaks down the complexity of the conversion but it is hard to imagine a large and complex relational database (RDB) fitting into a single HBase table without affecting its performance. HBase is, after all, a 'query first' type of schema design database. Another conversion proposal by Serrano et al. [2] proposed a conversion methodology in four steps. First, convert every one-to-one and one-to-many relationship into a single merged table in HBase. Second, apply a recursive method to merge neighboring tables. Third, design a row key, and fourth, create views for different access patterns needed. In this proposal, an additional step is performed that consisted in the extraction of access patterns from query logs. This last step should have been considered as a fifth step since access pattern extraction is a key step in the conversion process. To the best of our knowledge, there is no set of conversion rules to

convert between an existing RDB and HBase at this time and thus, the main contribution of this paper is to present a set of conversion rules that can be used for any conversion of an existing RDB into HBase.

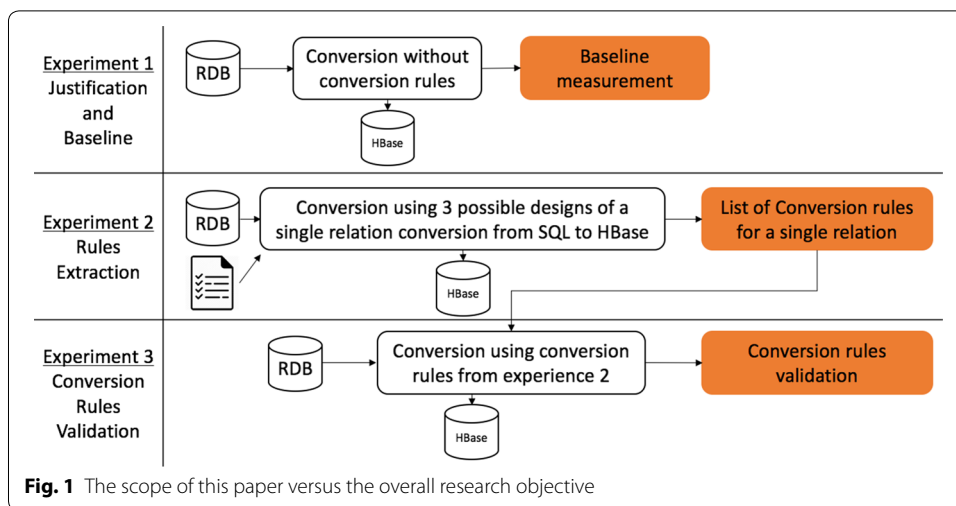
Because relational databases are based on mathematical theory, it is possible to convert any specific relational database implementation, such as a MySQL database for example, to any other relational database such as Oracle or MSSQL [3]. Recent NoSQL databases, on the other hand, like HBase for instance, are not relational and have a completely different schema design [4]. NoSQL databases are typically designed considering a specific use case: queries and access patterns rather than using a general relation and normalization process. This generates a difficulty for RDB-trained staff when the time comes to convert an existing legacy system based on RDB to NoSQL technologies. Database conversions are typically done using four steps [5]:

1. Schema translation
2. Data transformation
3. Data conversion
4. Data operations

Many tools and open source projects are already dealing with steps 2–4 [6]; consequently, this research focuses on the schema translation step. A schema translation uses an existing relational data model as an input and a non-relational (e.g. NoSQL) data model as an output. NoSQL databases are categorized by many types such as Wide Column Store/Column Families, Document Store, Key Value/Tuple Store among many others. For this research, the focus has been set on a Wide Column Store/Column Families category, more specifically the popular Hadoop database called HBase.

The main objective of this research is to propose a set of conversion rules to support and improve the schema conversion process of existing applications using RDB to one of the NoSQL technologies, the column oriented database named HBase. First, an experiment is conducted to justify the need for conversion rules by having participants convert an existing application to HBase without any guidelines. This first experiment served as a baseline for experiment 2. Then, another experiment is conducted to uncover a first list of conversion rules by having participants conduct a real conversion of an existing application to HBase using three possible conversion design patterns (mainly one-to-one and one-to-many relationships) as shown in Fig. 1.

This paper is structured as follows. “[HBase schema basics and design fundamentals](#)” presents the HBase schema basics and design fundamentals. “[Experiment description and results](#)” presents the first experiment with participants performing a conversion simply based on their experience, which tries to simulate how an individual would do such a conversion without any guidelines today. Next, three RDB to HBase conversion design patterns are presented and offered to be used in the second conversion experiment: a conversion with the same scenario as experiment 1, and conducted using three possible designs of a single relation conversion from RDB to HBase. “[Rules extraction](#)” presents how the conversion rules for the schema translation were extracted from the second experiment. Finally, “[Conclusions and future research](#)” presents the main conclusions and future improvements.



HBase schema basics and design fundamentals

This section presents some of the key terms and concepts as well as some design fundamentals about HBase technology, prior to introducing the experiments in “[Experiment description and results](#)”.

A RDB Schema is composed of tables, columns and relationships between the tables: one-to-one, one-to-many and many-to-many. Such relationships do not exist in an HBase schema [4]. The HBase terminology refers to a row key, a column and a column family. To facilitate the understanding of this paper, the term “HTable” will refer to an HBase Table and the term “Table” will refer to any relational database table.

Row key design in HBase

The selection of the row key in an HTable is one of the most critical activities for a successful schema design [7]. A row key is used to fetch data, which means that when a row key is carefully selected, its rows are sorted in a way that regroups the data to be accessed together, ensuring a more efficient query [7]. A badly chosen row key could spread the data across the HTable and make fetching of the data time consuming and result in poor performance. Therefore, the selection of a row key is always based on the context of the table, as opposed to the relational model, where the primary key plays this important role. Typically, the primary key is unique in a table but this does not usually allow for grouping of data. Grouping can be performed using SQL statements such as ORDER BY on data contained in other columns (e.g. SELECT * FROM BOOKS ORDER BY AUTHORS).

Columns and column family in HBase

One of the first comparisons that can be made when converting a table to an HTable is the use of columns. The notion of a column in an HBase table is quite different from that of a column in an SQL table. In an SQL table, a column is meant to hold a single piece of data type whereas a column, or rather a column family, in an HBase table can contain multiple pieces of data via a key-value pair. A column family in an HBase table should be considered as a container of key-value pairs. The column family is determined by the

information in each row and it regroups key-value pairs of related data. For example, the column family “address” contains the following columns: house number, street name, city, province, country and postal code. One of the major differences between columns in a RDB and an HBase column family is that there is no strict definition for HBase columns and they can be added dynamically when needed. Therefore, if the address is different from one country to another, new columns can be added to the column family when needed. This flexibility would be very hard to build into an address table using relational databases, as we would need to modify the table in the future if new columns were needed. The nature of a column family lends itself well to this type of situation since the HTable will not need modifications to accommodate the differences in addresses. However, this imposes a restriction on the conversion of an RDB table, as it cannot be faithfully recreated in NoSQL.

HBase design fundamentals

RDB use three types of relationships between tables: one-to-one, one-to-many and many-to-many. The first two are designed using one table containing a foreign key that references the second table. As for the many-to-many relationship, it requires a third intermediate table that holds references to the other two tables, which can ultimately be broken down into two one-to-many relationships between the first and third table and between the second and third table. Therefore, the many-to-many relationship is simply a combination of one-to-many relations between three tables, which also means that there is no direct link between the first and second table as opposed to the one-to-one and one-to-many relationships. Thus, for conversion purposes, we can only consider a relation that has a direct link between two tables.

In HBase, it is quite different since such relationships do not exist. Without relations, the focus when creating or converting an RDB schema to HBase is to define the access pattern upfront and ask the important questions (i.e. also known as “what is your use case”), namely what is accessed, how it gets accessed and when is it accessed. The following are well-known methods to convert a simple RDB relationship. It can be converted into either a one HTable or a two HTable design (i.e., HTable being an HBase table). A one HTable design results in two possible conversions: the first would be a merge of both RDB tables into one column family and the other would be to create two column families, one for each RDB table. The two HTable design conversion closely resembles its RDB table counterpart, as each HTable will hold one column family for each RDB table.

An HBase design schema can be summarized as follows:

1. Generally, all the data that are accessed at the same time should be grouped into a single column family to ensure that searching is efficient. Therefore, this would be represented in a single table containing a single column family. Otherwise, if the data being accessed that need to be close to each other are accessed at different points in time, we could represent this with a single table but with two column families.
2. If the data are related to each other but can be accessed independently, then two HBase tables having one column family each should be created.

3. Physically, column families are stored on a per-column family basis. Therefore, when accessing two column families in the same HTable, HBase will access two separate physical files.

Lastly, it is recommended to limit the number of column families within a table. Only introduce a second or third in the case where data access is usually column scoped.

Experiment description and results

Experiment 1

Experiment 1 goal

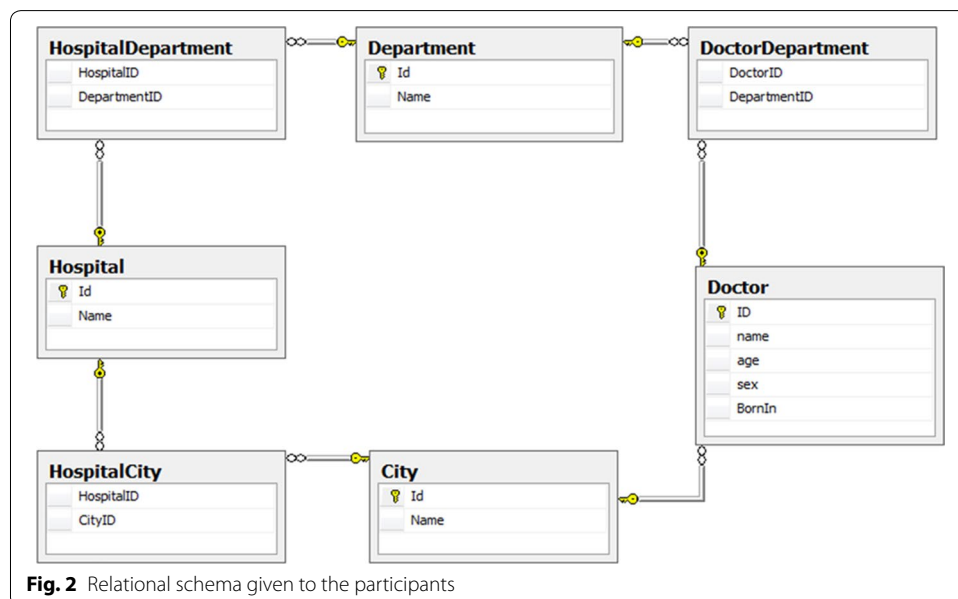
The first experiment was conducted to investigate the need for schema conversion rules by asking participants to convert an existing RDB database to HBase without any guidelines [8].

Experiment 1 description

This subsection describes the experimental design used according to the structure recommended by the following authors: Easterbrook et al. [9], Marcos [10], and Zelkowitz et al. [11]. This first experiment was conducted using a heuristic approach, i.e. the participants use their experience, educated guesses or plain common sense to do the conversion. The material provided to the participant for the execution of that experiment was:

- 1) A tutorial session and document;
- 2) Participant’s instructions guide;
- 3) A RDB schema to be converted (Fig. 2);
- 4) A survey questionnaire.

The tutorial session included a training document, which summarized the content of the tutorial session. These documents were distributed to each participant beforehand.



It included a summary of RDB and NoSQL concepts with practical examples of each. Then we conducted a 2 h tutorial, going through the document step by step and answering questions.

The participant guide included instructions, booklets of worksheets that the participant used during the experiment. One of these worksheets is entitled “The NoSQL solution” (e.g. the proposed HBase schema) where the participant was asked to draw a representation of his proposed HBase schema (i.e. the target schema) for the conversion.

Figure 2 shows the RDB schema that had to be converted to NoSQL by the participants. This schema is the one already used in Singh’s experiment [12] and is composed of seven relational tables, where:

- a. Four are large tables (e.g. City, Department, Doctor and Hospital), and
- b. Three are junction tables (e.g. DoctorDepartment, HospitalCity and HospitalDepartment).

The City, Department, Doctor and Hospital tables have an “Id, Name” structure, with “Id” as primary key. The “Doctor” table contains “Id, Name, Age, Sex and BornIn”. The latter field is the “Id” of the city where the doctor was born. The junction tables allow for the expression of the “many-to-many” relationships indicated by each junction table’s title. It is important to note that each participant was offered the opportunity to choose between several sub-schemas from the main schema. For instance, a participant could choose only to convert the sub-schema composed of the entities Hospital—HospitalDepartment—Department or the sub-schema Doctor—DoctorDepartment—Department or the participant could select the entire schema. Note the five key aspects of this RDB conversion that best represent the items that must undergo conversion: Tables, Constraints, Primary Keys (PK), Foreign Keys (FK) and other elements (others like; fields, types of relationships, views, indexes, procedures and triggers).

Finally, a survey form was given to each participant after the experiment. The survey was designed and validated following the recommendations of the following authors: Kasunic [13] and Lethbridge [14]. It contained 9 questions where the first four questions were oriented toward the “experience” of the participants:

1. First question is an academic background question with 3 possible answers: Graduate with Ph.D., Graduate with Master or Graduate, and Undergraduate Student;
2. The second question enquired about the working status and had only 2 possible responses: working in industry or currently in academia;
3. The third question collects data about the number of years of work experience using RDB technology with 4 possible answers: no experience, little or low level of experience (e.g. a few days to 1 year), average amount of experience (categorized as from 2 to 5 years), and advanced experience (e.g. very good knowledge with more than 5 years of use);
4. The fourth question asked if the participant had any NoSQL experience and captured their answer using the same 4 answer choices as above;
5. The fifth question concerned the conversion process and was especially focussed on reporting the first step that was used to initiate the conversion process;

6. The sixth question asked to report the amount of effort needed to perform the conversion (i.e. without the use of rules to guide the process). This question could be answered using values from 1 to 5, where 1 indicated that the process was easy to achieve without major effort, a value of 3 indicated that it was achieved using a large amount of effort and a value of 5 meant that no matter how much effort was put in, it was not possible for the participant to successfully perform the conversion;
7. The seventh question was designed to evaluate the level of confusion experienced by the participant during the conversion process (e.g. no idea where to start or what the next step was). This question had 5 possible answers: 1: always confused during the process, 2: very often confused, 3: sometimes confused, 4: rarely confused, and 5: never confused;
8. The eighth question was presented in the form of a matrix to assess the conversion percentage achieved of the proposed solution regarding each of the relational aspects mentioned earlier (e.g. Table, Constraint, PK, and FK);
9. The ninth and last question asks the participant if having access to conversion guidelines would have improved the conversion process. This question had five possible answers: 1: strongly agree, 2: agree, 3: undecided, 4: disagree, and 5: strongly disagree.

Finally, eighteen participants conducted the conversion experiment, filled out a NoSQL solution sheet and expressed their opinions by completing the survey.

Experiment 1: data and analysis

Table 1 presents the educational level of the participants who were mostly working in industry and enrolled part-time in software engineering university programs: 6% of participants were taking an undergraduate course and 89% of participants were taking graduate courses (i.e. 39% at the Bachelor level and 50% at the Master level).

Table 2 shows that 83% of the participants were currently working in industry: this is aligned with our goal in trying to simulate what actual software engineers, in the industry, would experience during such a conversion. It can also be observed, in Table 3, that

Table 1 Educational level of the participants (N = 18)

Educational level		
Classification	Response in percentage (%)	Response in number
Ph.D	5	1
Master	50	9
Bachelor	39	7
Undergraduate	6	1

Table 2 Work area of the participants

Work area	
Classification	Response in percentage (%)
Industry	83
Academic	17

a great number of participants had previous experience with RDB technology and that 45% of them had more than 5 years of experience with RDB technology.

As expected, Table 3 also shows that 94% of the participants had no previous knowledge of NoSQL database technology.

The experience profile of the participants indicates that a set of guidelines could be a valuable tool for practitioners in such conversion processes. Figure 3 also shows the different approaches taken by the participants for the first step of their conversion. Considering that most participants had industry experience, Fig. 3 indicates that 61% (i.e. 33 + 28%) of the participants chose to begin with the RDBMS “tables” element for the conversion.

The next criteria analyzed is the perceived difficulty encountered during this conversion process. Depicted in Fig. 4, the initial perception that this conversion is difficult was reported by 78% of the participants (i.e. 39 + 39%). This result is consistent with individuals who had very little exposure to NoSQL concepts beforehand (as reported in Table 3 level of experience in NoSQL).

The participants were of the opinion that this conversion process demanded a considerable amount of effort as shown in Fig. 4.

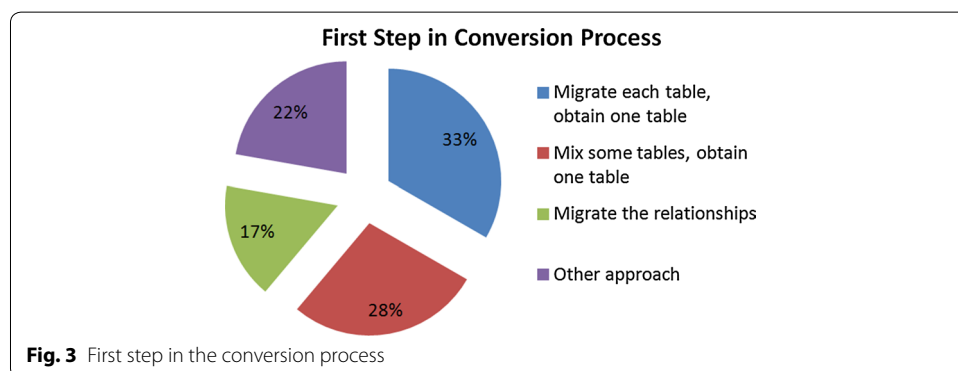
Finally, Fig. 5 evaluates the level of confusion experienced by the participants during this conversion process. It reports that the majority of the participants (56%) had felt sometimes or always confused, i.e. not knowing how to go about the process.

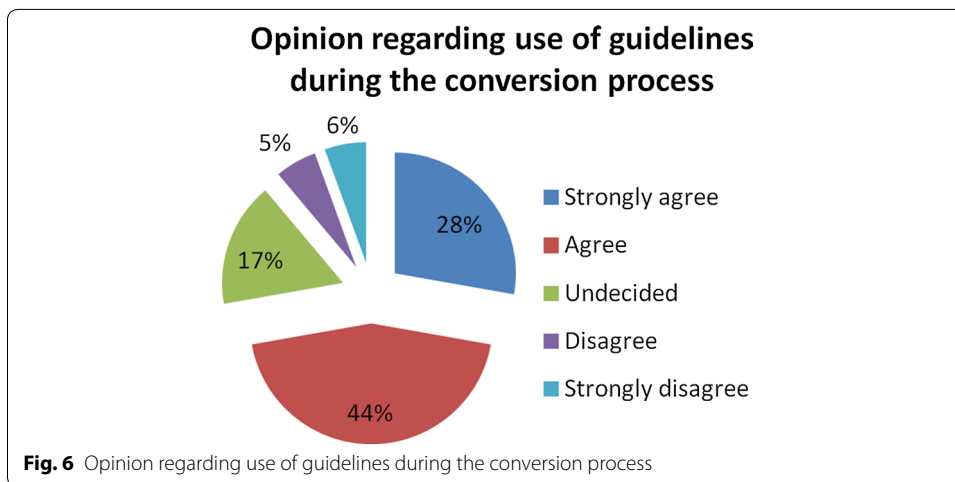
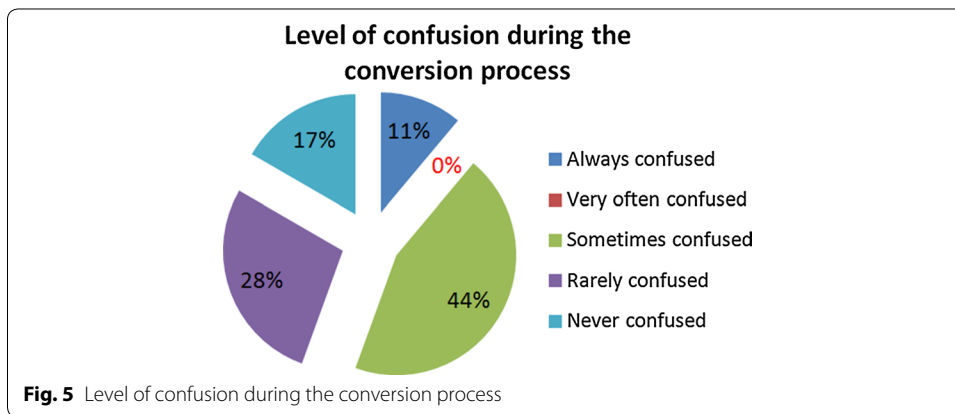
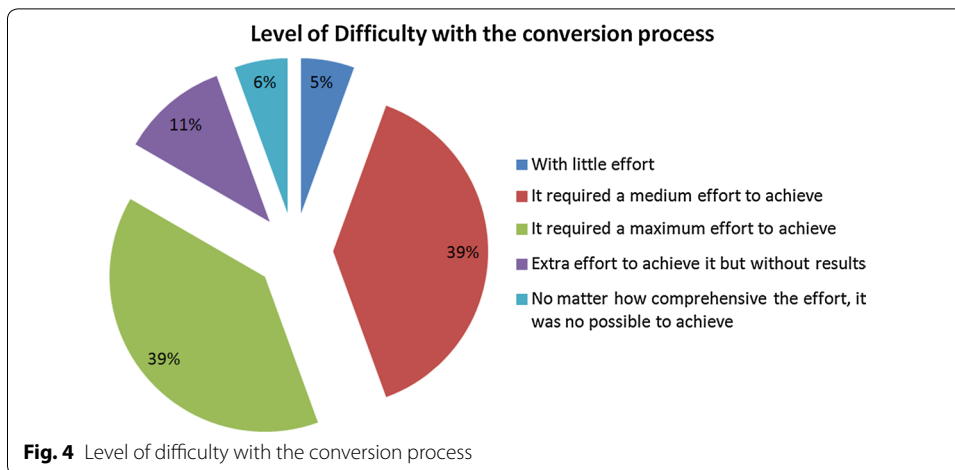
Figure 6 provides the opinion of the participants regarding whether the process would have been easier if a set of guidelines had been provided: 28% strongly agreed with their usefulness and 44% agreed with the relevance of this kind of tool to help in this conversion process. This was to be expected and was observed.

Concerning the five key aspects of RDB that were studied in this conversion experiment: Tables, Constraints, Primary Keys (PK), Foreign Keys (FK) and other elements (others like; fields, types of relationships, views, indexes, procedures

Table 3 Level of experience in DB

Type of DB	Experience in years			
	No exp	Low exp (<1 year)	Middle exp (2–5 years)	Advanced exp (>5 years)
RDB (%)	22	11	22	45
NoSQL (%)	94	0	6	0





and triggers), Table 4 summarizes, for each of these RDB aspects, the participants' reported percentage of coverage in intervals: 0–24, 25–49, 50–74, 75–99 and 100%.

Table 4 Level of coverage in different DB aspects

Relational aspect covered	Percentage of coverage				
	0%	25%	50%	75%	100%
Table (%)	22	0	6	22	50
Constraint (%)	39	11	17	5	28
PK (%)	29	0	18	12	41
FK (%)	28	0	11	22	39
Others (%)	94	0	0	0	6

For example: 50% of the participants reported that their conversion solution proposal adequately covered the relational aspect “Tables” by 100%. In contrast, 22% considered that their proposed solution did not cover this aspect at all, i.e., 0%. Moreover, Table 4 shows that 28% of the participants feel that their proposed solution covered 100% of the relational aspect “constraints”. On the other hand, 39% of the participants think that their solution did not cover this aspect at all (0%). Furthermore, Table 4 shows that 41% of the participants think that their proposed solution covers 100% the relational aspect PK. However, 29% report that their solution did not cover this aspect at all (0%). Table 4 also reveals that 39% of the participants believe their solution covers 100% the relational aspect FK. Conversely, 28% consider that their solution did not cover this aspect (0%). Other RDB elements aspects such as fields, store procedures or triggers were aggregated in the “others” category and 94% of the participants felt they had not covered these aspects fully during the conversion (see last row of Table 4).

The next sub-section discusses the second experiment that followed this first heuristic approach to the conversion.

Conversion design patterns

The approach selected to identify a preliminary list of conversion rules was to use a few samples of a relational schema and see how they could be converted into an HBase schema; the outcomes of this step will be called conversion designs patterns. This was prepared in our research lab by the authors before initiating a second experiment with the participants. This step aimed at identifying the best corresponding HBase schema for each sample relational schema and next, extract/generalize rules to be used in experiment 2 that could help participants with the schema conversion. Through a number of iterations, the following three conversion design patterns were identified.

One-to-one relationships

For the conversion of a one-to-one relationship, there are three possible ways (i.e. 1a, 1b or 2) to convert it into HBase:

1. *Create one HTable* The first pattern would be to merge the two RDB tables together into one HTable. The resulting HTable could have two resulting designs:
 - a. *Two-column family* One column family to store the first RDB table columns and a second column family to store the second RDB table columns.

- b. *Single column family* This design pattern is recommended due to HBase limitations [15].
2. *Create two HTables* The third possible design pattern is to create two HTables where each HTable contains one column family and each column family contains all RDB columns. Finally, insert the row key of each HTable into both HTables.

The question that remains is how to choose one design over another? This is determined primarily by the context of the data access. If the original intent is to save space to optimize the caching of pages in RDB, then this is no longer a consideration in HBase as the columns in the column family are dynamic and only added when needed. In this instance, merging two tables into one would be valid since saving space is no longer a concern in HBase given that null values are not saved.

An additional consideration would be whether there are two tables that contain information that becomes irrelevant when they are disconnected; for example, the relationship between a person and a passport as in Fig. 7. This relationship is of no value when considered individually as a Passport needs to be attached to a Person to be valid and a Person needs to be attached to a Passport. For the purpose of this example, only valid passport is considered.

As a result, a Person can only have one valid Passport. Two tables represent this relationship: one that holds Person information and the second that holds Passport information.

When converting this relationship into an HTable, the Person table would include another column family that would hold the passport information that belongs to a Person, thus reducing the two-table design in a relational database to one HTable as shown in Fig. 8.

The reason for the inclusion of the Passport number in the Person table is due to the context by which the information is usually accessed. To find the Passport number of an individual, a lookup of the Person is always required. A Passport number is meaningless without the Person information that accompanies it. In contrast, Person information is meaningful even without a Passport.

Another architectural decision to consider is whether to keep the design of two tables. This would happen if the secondary table contains information that is meaningful, in and of itself. Consider a relationship between an employee and a workstation and assume that an employee can only have one workstation. This relationship between the two tables should be represented in HBase as two tables as well. The reason for this is

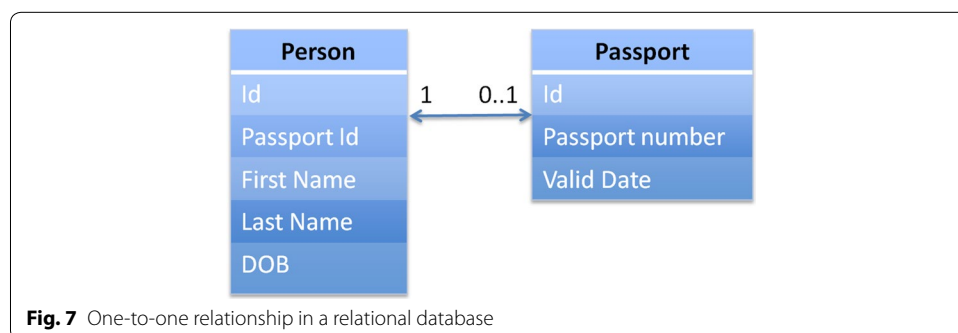


Fig. 7 One-to-one relationship in a relational database

Row Key	Column Families	
	Person:	Passport:
<LastnmFirstnm>	Person: Firstnm Person: Lastnm Person: DOB	Passport: Number Passport: Valid_date

Fig. 8 One HTable with two column families

again context dependent since information about the workstation is relevant even without needing to know the employee information. Suppose that the address of the computer changes, an access to workstation table would be required to update the address information. If the employee to whom the computer belongs is also a required information, it will put an unreasonable burden on the lookup of the location of the workstation. In addition, a one table design would assume that every workstation is assigned to an employee although a company can have unused workstations that are not assigned to anyone and therefore not entered in the database.

One-to-many relations

In the conversion of a one-to-many relationship, we found three possible ways to convert it into HBase:

1. *Create one HTable with two column families* This requires merging the information from the two related RDB tables. Each RDB table would be stored in a separate column family within the same HTable.
2. *Create one HTable with a single column family* This would entail that both RDB tables be merged into a single column family.
3. *Create two HTables* In this design, each RDB table would be added to a separate HTable with a single column family. The first HTable contains the source of the relationship; a second column family is added that contains the referenced Row Keys from the second HTable.

Once we had our conversion design patterns, we were ready to conduct the second experiment.

Experiment 2

Experiment 2 goal

In experiment 2, the participants were divided into groups with each group assigned a distinct HBase conversion design pattern presented in “[Conversion design patterns](#)”. The goal of assigning these conversion design patterns was to expose participants to different schema and thus avoid having the majority of students using a similar design by following a trial and error method. Another objective was to build a body of knowledge by analysing the impact of all possible schemas, good and bad. Finally, having many participants helps ensure that every design pattern will be used.

Experiment 2 use case description

A use case was developed for the experiment where participants were asked to convert a one-to-many relationship using the conversion patterns provided as presented in “[Conversion design patterns](#)”. To accomplish this second experiment, a case study had to be developed simulating real life situations and characteristics:

1. A large relational database currently experiencing performance issues and would benefit from a conversion to NoSQL technology;
2. An unknown industry domain to eliminate this factor in the experiment;
3. Find a population similar to experiment 1 to ensure valid and meaningful results.

A large RDB with such characteristics from the bioinformatics domain was available in our research lab. It originated from a hospital research laboratory where doctors and researchers were working on genomics and cancer research. This laboratory uses a RDB database that is facing major performance issues. An ÉTS student, Anna Klos, [16] had successfully solved this performance issue by converting the very large RDB tables to HBase.

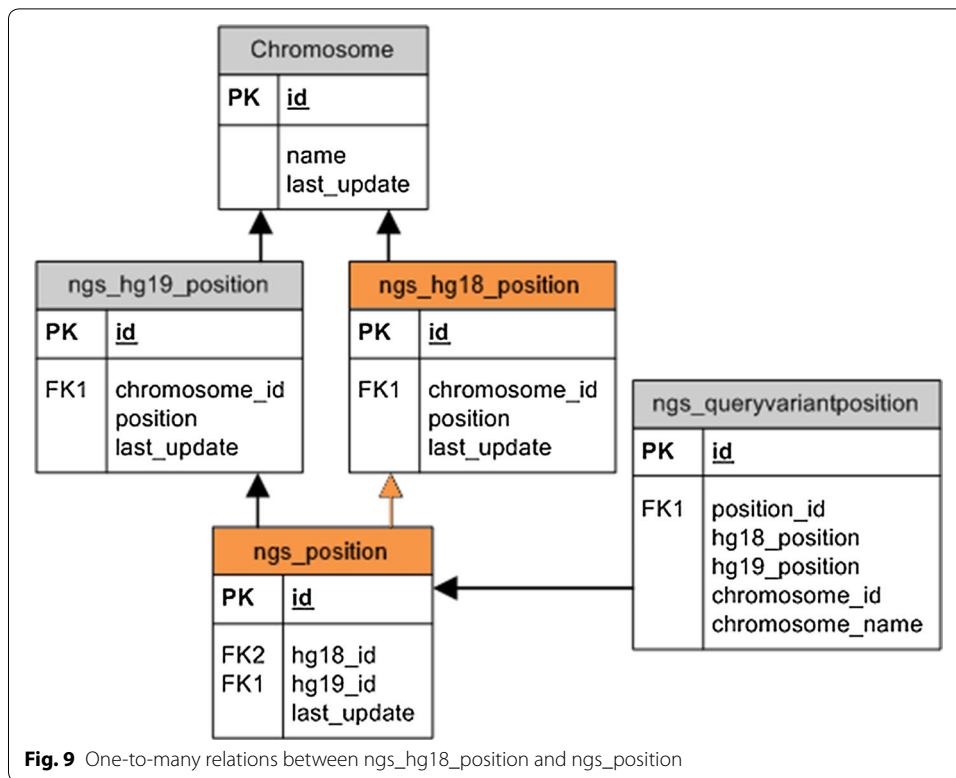
RDB for experiment 2 The genomic database mainly stores information about human chromosomes and focuses primarily on the chromosome relationship with diseases that affect the brain and nervous system. The researchers investigated 13 tables in this RDB to locate the performance issues. In this experiment, two specific tables were found to create most of these issues as shown in Fig. 9.

Participants This time, the participants were also ÉTS students, with different education and experience levels, enrolled in the undergraduate course “Introduction to Big Data”. The course objective was to introduce students to Big Data technologies including NoSQL databases. The class included 36 students that were divided into 9 teams.

Experimentation 2 The experiment was introduced in the form of practical class assignments. The three assignments were as follows:

1. *Assignment 1* First replicate a portion of this genomic database using the same RDB technology (participants were asked to use PostgreSQL) as shown in Fig. 9; then load the provided data (very large) and execute the SQL query provided by the hospital lab that experienced performance problems and capture execution time (see Fig. 10).
2. *Assignment 2* Convert the PostgreSQL database schema created in assignment 1 to HBase; then load the same data into HBase and convert the same SQL query that was earlier provided to generate the same results. This resulting query should typically be a scan function in HBase, as described in “[Experiment 2: data and analysis](#)” (Design 1).
3. *Assignment 3* Run the query created in assignment 2 on HBase; make sure it obtains the same results as previously in PostgreSQL and capture the execution time of the HBase implementation.

Each team was assigned a specific conversion design pattern as presented in “[Conversion design patterns](#)”. The overall experimentation 2 steps that were followed during this controlled experiment are described in Fig. 11.



```

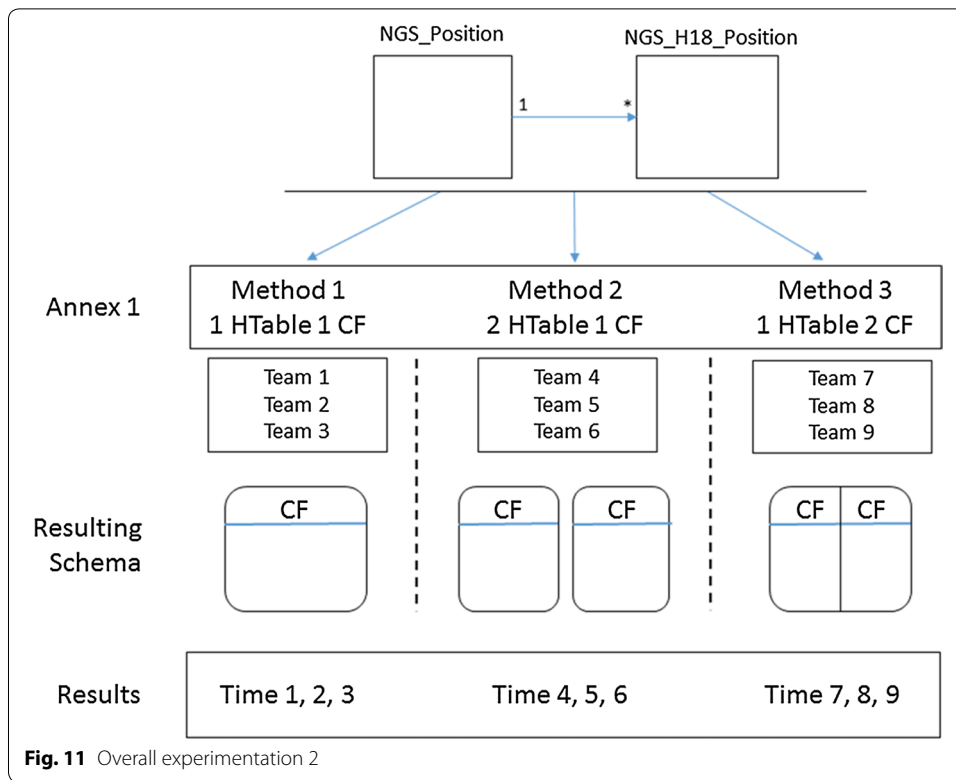
SELECT hg18.chromosome_id,
       hg18.position
FROM   ngs_hg18_position hg18,
       ngs_position p
WHERE  hg18.id = p.hg18_id
       AND p.hg18_id > 100000
       AND p.hg18_id < 1000000
       AND p.hg19_id > 0;
    
```

Fig. 10 sql ngs_hg18_position and ngs_position

The chosen relationship to be converted is located between table *ngs_hg18_position* and the table *ngs_position* as shown in Fig. 9. These tables were chosen because of their large data size (1.1 and 1.3 GB respectively). When ignoring the multiples queries related to a specific schema, it is difficult to predict the most efficient schema design. However, the query below was provided as one that is frequently run against these particular tables. The query includes a join between the two tables as well as a selection of rows based on some simple criteria as shown in Fig. 10.

Experiment 2: data and analysis

This subsection presents the data from the participants’ attempts to convert the SQL query (see Fig. 10) using the three conversion design patterns described in “Conversion design patterns”.



Design 1—one HTable and one column family Using the first schema of one HTable and one column family as shown in Fig. 12, the participants converted the data and created the query below, which returned the same data as the originating SQL query in Fig. 10

```
Scan 'ngs_hg18_and_position_t2_m1',
{COLUMNS => ['hg18_and_position_family:ngs_hg18_position_chromosome_id',
'hg18_and_position_family:ngs_hg18_position_id'], STARTROW => '000100000',
ENDROW=>'010000000', FILTER
=>"SingleColumnValueFilter('hg18_and_position_family','ngs_hg19_position_id',>,
'binary:0')"}
```

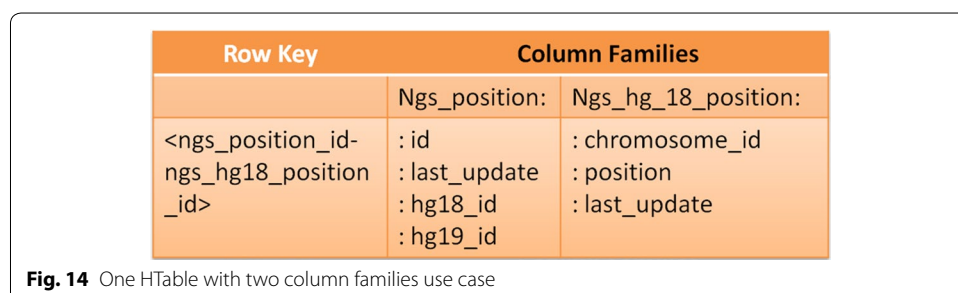
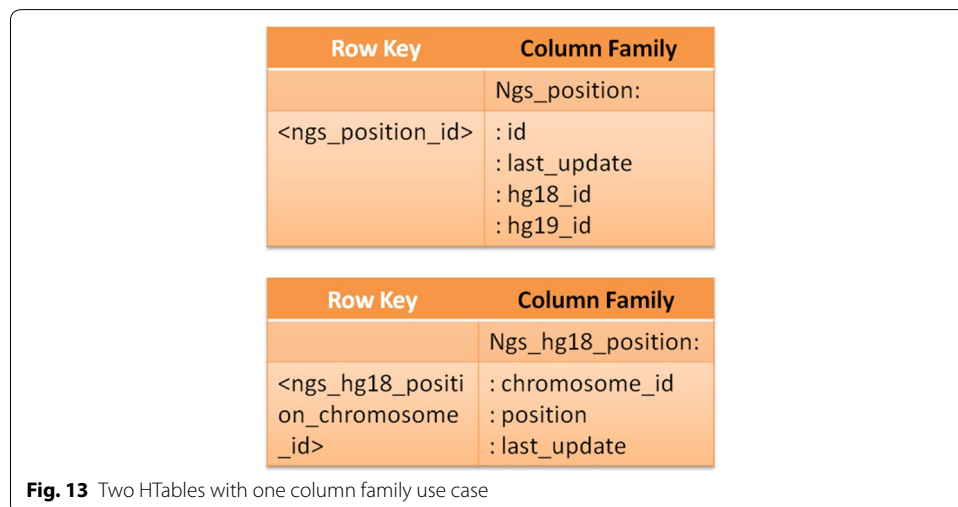
Row Key	Column Families
	Ngs_hg_18_position_and_nginx_position:
<hg18_id-position_id>	: chromosome_id : position : ngs_position_last_update : id : ngs_hg_18_position_last_update : hg18_id : hg19_id

Fig. 12 One HTable with two column families use case

The HBase scan that was executed yielded a query that returned 900,001 rows in 878 s.

Design 2—two HTable with one column family per table In the two HTable design shown in Fig. 13, data must be fetched from both HTables and then used in a comparison to filter out the wanted data to complete the SQL query requested in Fig. 10. A join clause between the two HTables is necessary to accomplish this filter. Since HBase does not use relations in its design [17], the concept of a query with a join is not possible. Therefore, none of the participants was able to find a matching query.

Design 3—one HTable with two column families The participants in this group created an HTable as shown in Fig. 14 and attempted to replicate the SQL query. The results obtained using the scan below, timed out after a long execution period. We should state here that the infrastructure used in the experiment has not yet been described. The distribution and parallel processing of HBase was not taken into account since the lab infrastructure for the experiment was a single virtual machine running on the same server as experiment 1 (i.e. attempting to replicate the previous PostgreSQL environment). This query would not time out if it was executed in a multi-server, distributed environment. Nevertheless, this approach performed poorly in comparison with other schema design patterns, as we will see




```
scan't2',{COLUMNS=>['t2_2:chromosome_id','t2_2:position','t2_1:
hg18_id','t2_1:hg19_id'],
FILTER=>FilterList.new([SingleColumnValueFilter.new(Bytes.toBytes('t2_1'),Bytes.toBytes('hg18_id'),
CompareFilter::CompareOp.valueOf('GREATER'),Bytes.toBytes('100000')),
SingleColumnValueFilter.new(Bytes.toBytes('t2_1'),Bytes.toBytes('hg18_id'),
CompareFilter::CompareOp.valueOf('LESS'),Bytes.toBytes('1000000')),
SingleColumnValueFilter.new(Bytes.toBytes('t2_1'),
Bytes.toBytes('hg19_id'),CompareFilter:
:CompareOp.valueOf('GREATER'),Bytes.toBytes('0'))]) }
```

Analysis of the results of this experiment

Design 1, which is “one HTable and one column family” yielded the requested data (of about 900 k rows) in 878 s. A disadvantage of using a single column family per HTable is that if too many columns are used, this will increase the size needed to store one row. Due to HBase distributed architecture, each column family is broken up into files of a specific size that split when the column family reaches the size limit [15]. This means that the larger the data stored in a column family of a row, the fewer rows there will be per column family file. This in turn makes the scan time longer as more files could potentially be needed to complete the query.

The reason why no group was able to fetch data when using two HTables is that joins are not inherently possible as in a traditional RDB approach. Since HBase is not a relational database, any relation between the information had to be made manually at the application level [18]. This translates into the need to write two queries. The first query would get the data from the *ngs_position* table and would use the returned result set as a filter with the second query on the *ngs_hg18_position* table. Therefore, this would call for extra logic to be added at the application level and would require more time to execute. Given that this SQL query is run very often, this would add extra time for the fetching of data.

For the third design, the issues with this pattern are related to the limitations in the design of HBase itself. In HBase, a column family is stored in a single file split across HDFS [15], which means that when accessing a row there is the potential of accessing two physical files. Considering the distributed nature of HBase, this implies that each column family file is not necessarily stored on the same server. Consequently, access time can be greatly increased if a scan needs to access both column families of the same table.

The result set that was obtained clearly points to the proper solution for this type of access pattern. Design 1 is the only situation that can provide meaningful results. Designs 2 and 3 were unable to yield any data, which demonstrates that the HBase schema is closely linked to the access patterns of the database. Therefore, this shows that when converting an RDB based application to HBase, an access pattern analysis is required to avoid proposing an HBase design that would potentially include weaknesses that would need further refactoring and costs.

Rules extraction

Using the knowledge gained from these two experiments and the three conversion design patterns experimented, we extracted considerations that should be taken into account when planning the conversion of an existing information system that uses RDB technology to the HBase technology.

Rule on data proximity

Firstly, every conversion will differ and depends heavily on the data access patterns currently used. Since HBase does not have any steadfast rules, it needs to have an understanding of how each data field relates to another. One of the most important rules that requires serious consideration is how the data will be accessed by the application. For that reason, the distance between frequently accessed data, in one scan, will definitely affect the query performance. Therefore, the data needs to be stored in the same column family, as much as possible, so as not to require multiple scans to obtain the desired information. This goes as far as maintaining a minimum amount of information pertaining to a secondary table and removing the need to go into the secondary table to get the information that is required. This rule can be inferred in the results of the two HTable design pattern of our use case. Since HBase has no implicit way of creating relationships when performing a scan [6, 10], the application itself will need to complete this action. This introduces extra complexity/effort that could be avoided by using a more optimal schema designed specifically for the data access patterns.

Rule 1.1: The more the information is accessed at one time, the closer the data needs to be stored. More specifically, when the same data is often accessed together, it should go into a one HTable design within the same column-family.

Rules on column families

The concept of RDB columns does not implicitly translate to a column family in HBase. As presented above, having many column families in one HTable can be detrimental to the schema translation due to the physical design limitations [15] of HBase. Since each column family is stored in its own separate file and then stored in a distributed manner, the larger the distribution, the less likely that the physical column family files will be stored in the same region server [15]. Thus, the second rule inferred from these experiments is not to store information in column families that are of vastly different sizes [15]. As a column family is broken up into different files and distributed in the database, the larger each row is, the less number of rows will be stored within the same file and therein creates the need for more files. This in turn will require the scan to access even more files to fetch the information wanted. However, the larger the network, the less impact this will have as the information will be read on each node and aggregated together by the master [15].

One major concern when designing an HTable is that the information will be distributed as evenly as possible. If there is a disparity in the number of rows needed to store information between two column families, this could lead to longer scans of the smaller column family as there are fewer records in the column and therefore, more empty space between rows [19].

Rule 2.1: The number of column families defined in each row should be no more than 2 or 3; otherwise, the performance will degrade due to the physical design of HBase. Specifically, because flushing and compression are done on a per region basis, if one column family is carrying the bulk of the data bringing on flushes, the adjacent families will also be flushed though the amount of data they carry is small. With many column families, the flushing and compression interaction can make for a number of needless I/O loading [15].

Rule 2.2: The number of rows in each column family needs to be roughly the same; then the columns should be split and references placed in each column family to allow data to be linked.

Rules on data quantity

HBase is designed to handle large amounts of data. HBase was modelled on Google's own BigTable [20] for that purpose. It has incorporated the ability to handle large data requests and by doing so, also incorporated some physical limitations on how the data is stored. These limitations need to be considered when converting an RDB schema as the expected performance gains may be mitigated given the limitations. The limitations impose two design considerations. The first is with respect to the size of the column families in a single row. Each column family needs to be of relatively similar size as they are stored in separate physical files of a maximum defined size. Once the maximum file size is reached, it is split and then dispersed. This means that when data stored in one column family is significantly larger than the other, it will create a situation where fewer rows are stored per physical file and again cause longer scans [21]. The second consideration is that column families should generally have the same access pattern. Otherwise, when fetching data, the scan could take much longer if the access pattern to each column family is vastly different.

Rule 3.1: Data size between columns families, in one row, need to be of similar size, otherwise, the column family needs to be split into two separate HTables, each containing a column family.

Rule 3.2: Column families in the same HTable should generally have the same access patterns.

Rules on access patterns

The context in which the data is accessed in the application defines the access patterns and the accessibility requirements. These will be the foundation upon which the HBase database will be created. As shown in “[Experiment 2: data and analysis](#)” (Design 1), this conversion was able to fulfil the required access pattern defined by the use case. Understanding the data access patterns and the data flow models of the application will assist in the design, especially since there are no steadfast rules when converting to HBase. The resulting schema will depend heavily on the context in which the data is accessed.

Subsequently, an analysis of the most used access patterns and of the heavy queries regularly performed is also needed before attempting a conversion.

Rule 4.1: Access patterns will define the conversion and need to be defined and understood; an analysis is needed before a conversion.

Rule 4.2: Analysis of heavy queries is required to obtain a proper design.

Conclusions and future research

This paper has identified and explored a set of rules to assist in the conversion of a relational database to one type of NoSQL database—the column oriented database named HBase. It describes a first experiment designed to evaluate if there was a need for a set of conversion rules. In this experiment, it was demonstrated that not all RDB practitioners could easily carry out such a conversion. Next, a second experiment showed how the

replicated query reacted depending on the design pattern used. This second experiment provided an opportunity to uncover a first set of schema translation rules for this particular conversion. A future step will explore a conversion experiment using these conversion rules with the goal to expand this initial body of knowledge and further validate the conversion rules, patterns and guidelines.

Authors' contributions

RO and AG conducted the experimentation under the supervision of AAp. JMD and AAb supervised and contributed to the writing of this article along with following each step of the experimentation and conclusion. All authors read and approved the final manuscript.

Authors' information

Rafik Ouanouki is a Senior Data Analyst at Laurentien Bank of Canada. He received a M.Sc. degree in computer science from the University of Montreal, Canada, in 2006. He is currently a Ph.D. Candidate at the ETS Engineering University in Montreal, Canada. His current research focuses on the conversion of existing SQL based software systems to recent cloud and BigData technologies. Rafik is a Canadian representative on the ISO Cloud Computing standardization committee: ISO/JTC/SC38.

Alain April received a Ph.D. degree from Magdeburg University, Germany in 2005. He is a professor of Software Engineering at ÉTS University in Montreal, and has published many software maintenance and SQA books. He is also the recipient of the 2011 ISO Award for Higher Education in Standardization. His current research interest includes Big Data applications for the banking, construction and health industries. Dr. April is a volunteer and active member of ISO and the IEEE Computer Society.

Jean-Marc Desharnais is currently adjunct professor at ETS University. He has a master's degree in public administration (ENAP) and in software engineering as well as a Ph.D. in Cognitive Science (UQAM). With over 35 years of experience in software engineering, he has been a Canadian delegate to the ISO SC/7 WG-6 (software quality measurement) since 2004. For the past 12 years, he has lectured at ÉTS (Canada), at the Middle East Technical University (METU) in Ankara and at the Bogaziçi University in Istanbul (Turkey).

Alain Abran is a Professor at the École de Technologie Supérieure (ÉTS)-Université du Québec. He was the Co-executive editor of the Guide to the Software Engineering Body of Knowledge project. He is also actively involved in international software engineering standards and is Chair of the Common Software Measurement International Consortium (COSMIC).

Abraham Gomez is Support & Implementation Specialist at TELUS Health Solutions, also is a researcher in computer science and a software developer with extensive experience in Java technology. His research interests and publications include artificial intelligence and cloud computing applications, focusing on data migration from relational environments to No-SQL environments. He has a M.Sc. in Computer Sciences and currently he is a Ph.D. Candidate at the ETS Engineering University in Montreal, Canada. Since 2011 he has been member of ISO/JTC/SC38, the ISO group committee, which objective is developing a standard for cloud computing.

Acknowledgements

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Consent for publication

Authors consent the right to publish this article by SpringerOpen.

Ethics approval and consent to participate

The project was accepted by the ethics committee of the ÉTS university on 10/11/2014.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 27 December 2016 Accepted: 22 March 2017

Published online: 04 April 2017

References

1. Lee C-H, et al. Automatic SQL-to-NoSQL schema transformation over the MySQL and HBase databases. IEEE international conference on consumer electronics-Taiwan (ICCE-TW), 6–8 June 2015; 2015. p. 426–7.
2. Serrano D, et al. From relations to multi-dimensional maps: towards an SQL-to-HBase transformation methodology. IEEE 8th international conference on cloud computing; 2015. p. 81–9.
3. Li N, et al. Database conversion based on relationship schema mapping. International conference on internet technology and applications; 2011. p. 1–5.
4. Lars G. The Apache HBase reference guide, version 2.0.0.0-snapshot, section 35 on Rowkey Design; 2013. <http://hbase.apache.org/book.html#schema>. Accessed 11 Mar 2017.

5. Maatuk A, et al. Relational database migration: a perspective. 19th international conference on database and expert systems applications. In: Bhowmick SS, Küng J, Wagner R, editors. Database and expert systems applications. DEXA 2008. Lecture Notes in Computer Science, vol 5181. Berlin: Springer; 2008. p. 676–83.
6. Chattopadhyay B, et al. Tenzing A SQL implementation on the Map Reduce framework. Proc Int J Very Large Data Bases (VLDB); 2011. p. 1318–27. <http://www.vldb.org/pvldb/vol4/p1318-chattopadhyay.pdf>. Accessed 11 Mar 2017.
7. Lars G. The Apache HBase reference guide, version 2.0.0.0-snapshot. 2013. <http://hbase.apache.org/book/rowkey.design.html>. Accessed on 11 Mar 2017.
8. Gomez A, et al. Building an experiment baseline in migration process from SQL database to column oriented No-SQL databases. J Inform Technol Softw Eng. 2014;4(2):1–7.
9. Easterbrook S, et al. Selecting empirical methods for software engineering research guide to advanced empirical software engineering. In: Shull F, Singer J, Sjøberg D, editors. guide to advanced empirical software engineering, Chapter 11. Berlin: Springer; 2008. p. 285–311.
10. Marcos E. Software engineering research versus software development. SIGSOFT Softw Eng Notes. 2005;30(4):1–7.
11. Zekowitz MV, et al. Experimental validation of new software technology, in Lecture notes on empirical software engineering. River Edge: World Scientific Publishing Co.; 2003. p. 229–63.
12. Singh P. Schema guidelines & case studies; 2010.
13. Kasunic M. Designing an effective survey, report number CMU/SEI-2005-HB-004. Pittsburgh: Carnegie Mellon Software Engineering Institute; 2005.
14. Lethbridge TC. A survey of the relevance of computer science and software engineering education. Proceedings of the 11th conference on software engineering education and training, IEEE computer society; 1998. p. 56–66.
15. Lars G. The Apache HBase reference guide, version 2.0.0.0-snapshot, section 24 on column family; 2013. <http://hbase.apache.org/book/columnfamily.html>. Accessed 11 Mar 2017.
16. Klos A. Optimisation de recherche grâce à Hbase sous Hadoop, technical report (in French), ETS University; 2012. http://publicationslist.org/data/a.april/ref-312/KlosAnna_LOG792_H12_Rapport_v_1-1.0.pdf. Accessed 11 Mar 2017.
17. Lars G. The Apache HBase reference guide, version 2.0.0.0-snapshot, section 140 on capacity planning and region sizing; 2013. <http://hbase.apache.org/book.html#ops.capacity>. Accessed 11 Mar 2017.
18. Lars G. The Apache HBase reference guide; 2013. http://hbase.apache.org/book.html#_joins.
19. Li C. Transforming relational database into HBase: a case study. IEEE international conference on software engineering and service sciences, Beijing; 2010. p. 683–7.
20. Chang F, et al. BigTable: a distributed storage system for structured data. OSDI'06, 7th symposium on operating systems design and implementation, Seattle; 2006. p. 1–16. <https://research.google.com/archive/bigtable.html>. Accessed 11 Mar 2017.
21. Lars G. The Apache HBase reference guide, version 2.0.0.0-snapshot, section 70 on regions; 2013. <http://hbase.apache.org/book.html#regions.arch>. Accessed 11 Mar 2017.

Submit your manuscript to a SpringerOpen[®] journal and benefit from:

- Convenient online submission
- Rigorous peer review
- Immediate publication on acceptance
- Open access: articles freely available online
- High visibility within the field
- Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com
