2015 IEEE First International Conference on Big Data Computing Service and Applications

An Evolutionary Algorithm for column family schema optimization in HBase

Fangzhou Yang, Jian Cao
Department of CSE
Shanghai Jiaotong University
Shanghai, China
{lake_titicaca, cao-jian}@sjtu.edu.cn

Dragan Milosevic
ZANOX AG
Berlin, Germany
dragan.milosevic@zanox.com

***Abstract*—Apache HBase is a column-oriented NoSQL key-value store built on top of the Hadoop distributed file-system. Logically, columns in HBase are grouped into column families. Physically, all columns in one column family are stored in the same set of files. Therefore the division of column families is closely related to the response time for a specific row query. In this paper, one new Evolutionary Algorithm is designed and applied to find the optimum column family schema for the given user queries. The reading performance of the optimized column family schema is evaluated on a real dataset provided by ZANOX AG, which contains 2.6 million rows of aggregated tracking data and 1.3 million user queries. It is shown that by using the found optimized column family schema, the reading performance of HBase is improved with a statistical significance. User queries from a testing set show that the average response time is reduced by up to 72% compared to un-optimized column family schemas.**

Keywords—HBase; NoSQL; Column Family; Column Layout; Schema Optimization; Evolutionary Algorithm;

I. INTRODUCTION

As the age of big data comes, more and more organizations became focused on delivering more targeted information, such as recommendations or online ads. Typically a great amount of data needs to be efficiently processed. For example, as the leading performance affiliate network in Europe, ZANOX AG tracks more than 1 billion transactions daily, which are created by users seeing and clicking on internet advertising. All those transactions need to be stored, aggregated and analyzed, and as the amount of the data continues to increase, the traditional relational database management systems (RDBMS) [1] cannot satisfy the requirement for online analytical processing (OLAP) [2], to answer multi-dimensional analytical (MDA) queries swiftly. To overcome this problem, a new scalable and distributed database different from the RDBMS is proposed. A NoSQL database [3] provides a mechanism for storage and retrieval of data using looser consistency models than RDBMS. These models feature the simplicity of design, horizontal scaling and a finer control over availability. While they have no fixed schema and no joins, they provide auto-sharding, a way of "scaling out", to automatically spread data across several servers.

Apache HBase [4], an important member of the NoSQL family, is an open-source, distributed, versioned, fault-tolerant column-oriented store modeled after Google's Bigtable [5]. Besides, Apache HBase is a consistency- and partition-tolerant distributed system according to the CAP theorem [6], which enables the execution of low latency OLAP queries at the storage side. Nowadays, HBase is successfully applied in many big companies, including Facebook [7], Twitter [8] and Yahoo [9].

In order to have a good performance, an HBase schema should be optimized on a use case level. On the one hand, row-keys of tables should be well-designed, because row keys are the only index of the table that determines how the data will be searched and scanned. There are a lot of successful row-key design instances for different use cases. For example, OpenTSDB [10] provides an efficient row-key design for time series based data, RichRelevance [11] utilizes row-keys that represent rules for capturing user behavior, and WibiData [12] carefully optimizes row-keys in order to support efficient data drilling. On the other hand, even in the high-dimensional space, typical OLAP queries require only a very small subset of columns [2]. Consequently, columns in the table should be grouped into column families (CFs) in order to make that a data access is column scoped. Since HBase is column-oriented, data is stored and sorted in CFs. If a scan operation requests only a subset of columns, the CFs without requested columns can be entirely omitted during the expensive scan operation and corresponding storage files will not be read at all. This is the power of the column-oriented architecture where it shines best.

In this paper, a novel approach to optimize the CF schema in HBase is proposed. It learns from past user queries, so that it can align the schema of CFs with the actual queries. As the result, the reading performance of HBase will be significantly improved. Unlike the row-key design that is tailored to very specific use cases, the proposed approach for the optimization of CFs is automatic, broadly applicable, easily extensible and useful even for other column-oriented databases.

The organization of this paper is as follows. Fundamentals and Related work are discussed in Section II. The challenges of CF schema optimization and the proposed evolution-based optimization algorithm are described in details in Section III and IV respectively. The Experiments and Performance Tests are presented in Section V. Finally, Section VI summarizes the paper and suggests some of the promising directions for future work.

978-1-4799-8128-1/15 $31.00 © 2015 IEEE
DOI 10.1109/BigDataService.2015.20

II. Fundamental and Related Work

A. HBase Schema Design and Optimization

HBase schema consists of two parts: the design of row-keys and the partition of columns into CFs. In order to have a well performing HBase system, both aspects are to be addressed.

1) Row-key Design in HBase

Row-keys are the only index in HBase table, which determines the lexicographical order of rows in regions of a table. Typically, a row-key is built of several different fields, which are as efficiently as possible concatenated together, where the order in which they are put together determines the distribution of the rows.

Different row-key structures are designed for different use cases. On the one hand, to achieve good writing performance, continuously written rows should be dispersed and scattered in different regions. As a result, data is written to different regions in parallel and that guarantees good writing concurrency. On the other hand, for good reading performance, the rows, which are requested by the scan operation, should be stored as close as possible, so that the data can be read as sequentially as possible [4]. As in OLAP, the reading performance has always a priority over writing, row-keys are designed to optimize scan operations.

2) Columns and Column Families

Since HBase is a column-oriented data store, columns and the CF schema are especially important when designing HBase tables.

Several optimization approaches for columns are possible. Similarly to the design of row-key, the name for column qualifiers and CFs should also be kept as short as possible for schema optimization. Additionally several columns can be compacted together to reduce the amount of space needed. For example, in OpenTSDB [10] all data points are squashed together and the amount of overhead consumed by disparate data points is reduced significantly. Data is at first written to individual columns for speed, and then is compacted later for storage/retrieval efficiency. Once a row is compacted, the individual data points are deleted.

There is practically no limit on the number of columns for one CF. It can be hundreds or thousands, and it will not cause any side effects. However, the number of CFs in an HBase table is severely restricted [4]. If one CF is marked for flushing and starts to flush data from MemStore into a Hadoop file-system (HDFS), adjacent CFs will also be flushed, even when those CFs only have a small amount of data. The flushing of many CFs will therefore cause additional, unnecessary IO load, many small files in HDFS and consequently the performance will decrease.

It is therefore suggested to try to use only one CF if possible in HBase schemas [4]. However, if everything is stored in one CF, then we cannot take advantage of the benefits of column-orientated storage. Therefore, an optimized CF schema, which matches queries and has an acceptable size (usually not more than half of a dozen), is required for optimizing HBase schema.

B. Schema Optimization on Column Layout

There are many studies on schema optimization of column layouts. A. Jindal et al. [13] categorize and compare the vertical partitioning algorithms across several dimensions based on the way they aim to solve the vertical partitioning problem. On the one side, Navathe partitioning algorithm [14] is one of the earliest approximation-based approaches to vertical partitioning. This algorithm is a top-down algorithm, which clusters attributes according to an attribute affinity matrix, and then splits them into vertical partitions recursively. O2P algorithm [15] is another top-down algorithm, which focuses on real-time partitioning. The basic idea of constructing and maintaining an affinity matrix is similar to Navathe algorithm. However, the computation and clustering in the affinity matrix is dynamic for each incoming query. On the other side, HillClimb algorithm [16] is a bottom-up algorithm, which iteratively finds two small partitions, whose merged partition provides the best improvement in terms of query costs. HYRISE algorithm [17] is a multi-level algorithm, which computes vertical partitions for main-memory resident data in order to minimize cache misses. The algorithm starts with the set of atomic partitions that are iteratively merged until query costs are reduced. Trojan layout algorithm [18] is threshold-pruning algorithm that creates vertical partitions for big-data. It uses different partition layouts for different data replicas and tries to optimize each one for different subclass of queries.

The above introduced vertical partitioning algorithms have been proposed under different settings for different scenarios. Vertical partitioning for column layout is quite similar to constructing optimal CF schemas in HBase. However, HBase is not a column-oriented database in the typical RDBMS sense [4], but a distributed database system that only utilizes an on-disk column storage format. Consequently, additional factors have to be considered for constructing CF schemas in contrast to general vertical partitioning on column layout.

III. Problem Description

HBase utilizes an on-disk column storage format, where columns are grouped into CFs. Physically, all columns of one CF are sorted and stored together as key-values in HDFS, and thus the division of CFs is closely related to the response time for a specific row query. On the one hand, if a user query is requesting only columns that can be found in one CF, all store files that correspond to other CFs can be completely omitted. As a result much less data is loaded and query will be processed faster. On the other hand, if requested columns are spread across several CFs, all the store files that belong to those CFs will be searched. As a result many more files need to be opened, more data is loaded and response time increases. Therefore, the optimized schema of CFs, which minimizes the number of CFs to be used, has a potential to significantly reduce the processing time for a specific query.

In real world applications, there are usually hundreds or even thousands of columns. Therefore, the problem of building the schema of CFs in HBase is of importance. Two solutions are commonly applied for building CFs. The simplest one is to give up on using multiple CFs and store everything into one

CF, which may lead to a high query time to search column items in large sets. The other one is to divide columns according to their types or semantic groups, which may also cause a high total seek time if some queries need to load data from different CFs at the same time.

Both solutions have their drawbacks. What we want to achieve, is to provide a solution to find a schema of CFs, in which the number of required CFs for a specific query set are minimized and data is evenly distributed across different CFs.

The data should be evenly distributed across different CFs because of the cardinality of CFs: For instance, if CF A has 1 million rows and CF B has 1 billion rows, CF A's data will likely be spread across many regions. This makes mass scans for CF A less efficient.

In addition, considering HBase is based on HDFS, which is a scalable distributed file system and can support terabytes of data with redundancy [19], we can sacrifice some additional space to duplicate columns in different CFs so that the number of required CFs for a given query-set can be further decreased. Besides that, a load balance across different CFs should also be considered during the construction of CFs, which means different requests will be evenly distributed across all the CFs instead of concentrating on a small subset. A load-balanced structure can make HBase Servers more robust and efficient.

In summary, the problem of building CFs for HBase can be described as the following complex optimization problem:

1. The number of required CFs should be minimized for a specific query-set.
2. Data of different columns should be evenly distributed across different CFs.
3. The size of duplicate columns should be minimized.
4. The number of CFs should be maximized (cannot be too large, because too many CFs are not suggested in the current version of HBase [4]).
5. The difference of load frequencies between different CFs should be minimized.

However, the optimization problem to find a best group of CFs is an NP Hard problem [20]. Assume we have 100 columns to form 4 CFs, there will be 2^{400} combinations in total. It is therefore impossible to enumerate all combinations to find the best one. For this hard problem, one promising solution is Evolutionary Algorithm (EA) [21], which can get an approximate optimal solution in acceptable compute time. Falkenauer [22] has devised an EA named "grouping genetic algorithm", which provides a good solution for the grouping problem. The aim of the grouping problem is to find a good partition of a set, or to group together the members of the set, which is quite similar to our problem of grouping columns into CFs. However, in our case, the problem is much more complex as one column can be duplicated into multiple CFs to better match queries.

In this paper, we are going to solve this complex optimization problem by designing a new evolutionary algorithm and evaluating the resulting schema of CFs with statistical methods. We want to show that the reading performance of HBase can be improved with statistical significance [23].

IV. Approach

A. Genetic Algorithm

Genetic Algorithms (GAs) [21] are stochastic, population-based search and optimization algorithms inspired by the process of natural selection and genetics [24][25][26]. Unlike other optimization approaches, which operate on a single solution at a time, GAs simultaneously work with several solutions forming a distinct population. This gives GA the ability of exploring solutions in different regions of a solution space concurrently, thereby exhibiting enhanced performance.

Fig. 1 is the flowchart of the simple GA. The initial population is created either at random or by using domain knowledge. The quality of solutions is evaluated by fitness function in the selection phase, which aims to select good solutions to be candidates that form a new population for the next evolution cycle. In the recombination phase, genetic operators are applied to the current population to generate or evolve new offspring solutions. There are two basic genetic operators: crossover and mutation. While a crossover operator combines two parental solutions to create an offspring based on a crossover probability, a mutation operator acts by slightly altering a selected solution in order to help escape local optimums.

Selection and recombination control the evolution process by striking a balance between exploitation (of selection) and exploration (of recombination), which makes GAs capable of effectively searching the solution space.

B. Algorithm Design

The in-built ability of GAs to explore solutions in the solution space effectively, together with an appropriate representation of solutions, domain-aware fitness function, a proper selection method and customized recombination operators made the cornerstone of the proposed EA for finding the optimal HBase schema. The following subsections shows how the algorithm is designed phase by phase.

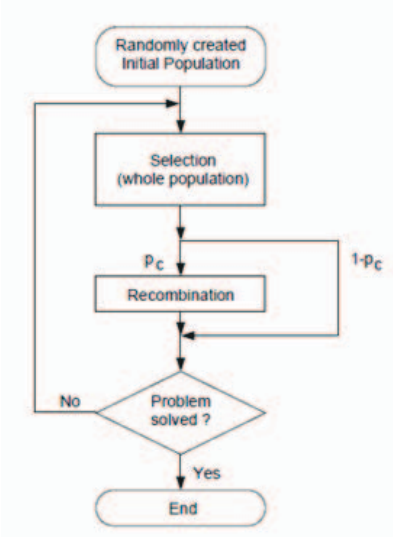

Fig. 1. Flowchart of Genetic Algorithms

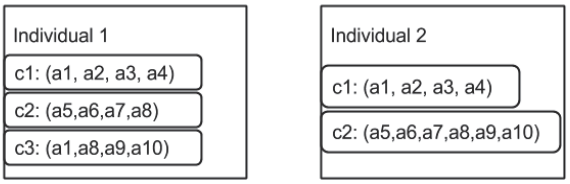

Fig. 2. Encoding example for CF schemas

1) Representation (Encoding)

A solution is a CF schema, which contains several CFs, and in each CF there are several columns. Unlike the classical GA, which encodes each solution as one chromosome, multi-chromosomes are applied to represent one CF schema solution. One individual contains several chromosomes, where one individual denotes one CF schema solution, one chromosome corresponds to one CF, and one gene represents one column.

Fig. 2 is an encoding example for two different CF schemas, where Individual 1 requires three chromosomes and Individual 2 needs two chromosomes in representation. More precisely, there are ten columns named from a1 to a10, which need to be assigned in different CFs. Solution 1 represented by Individual 1 has three CFs, where first contains a1, a2, a3 and a4, second a5, a6, a7, a8, and third a1, a8, a9, a10 columns. It is important to note that columns a1 and a8 are intentionally present in two CFs. As mentioned in section III, duplicate columns are allowed across different CFs to further improve the query performance. Individual 2 has only two CFs and no duplicated columns.

Not all individuals that can be represented are valid. A correct CF schema has to contain all given columns. Besides, a CF cannot contain duplicate columns due to the architecture of HBase. In short, one solution is valid if and only if:

1. Each column appears in at least one CF.
2. There are no duplicate columns inside one CF.

Validity of solutions should be guaranteed. Invalid solutions should be avoided and removed during evolution process.

2) Fitness Function Design

The fitness function evaluates the quality of every single solution. According to the multi-optimization problem as described in section II, the following five factors are defined respectively:

1. Consume Factor C describes how many CFs are needed for a given request (2).
2. Skew Factor S_k is measured by the variance of the number of columns in different CFs of an individual (3).
3. Duplicate Factor D describes the duplicate rate of columns of an individual (4).
4. Scatter Factor S_c is described by the number of CFs in a division solution (5).
5. Load balance Factor L_b, it is described by the variance of the loading frequency of different CFs of an individual for a given set of queries (6).

$$\mathrm{E} = \mathrm{W}_1\, C + \mathrm{W}_2\, S_k + \mathrm{W}_3\, D + W_4\, S_c + W_5\, L_b \tag{1}$$

$$\mathrm{C} = \frac{\sum_{\mathrm{n}} c_i}{m\, n}\ ,\quad \mathrm{C} \in \left[\frac{1}{\mathrm{m}}, 1\right] \tag{2}$$

$$\mathrm{S_k} = 2\sqrt{\frac{\sum_f \left(S_i - \frac{\sum_f s_i}{f}\right)^2}{f}}\ ,\qquad S_k \in [0,1] \tag{3}$$

$$\mathrm{D} = \frac{\sum_{\mathrm{n}} a_i}{m\, n_{atts}}\ ,\quad \mathrm{D} \in \left[\frac{1}{\mathrm{m}}, 1\right] \tag{4}$$

$$\mathrm{S_c} = \frac{1}{f}\ ,\quad S_c \in \left[\frac{1}{m}, 1\right] \tag{5}$$

$$L_b = 2\sqrt{\frac{\sum_f \left(l_i - \frac{\sum_f l_i}{f}\right)^2}{f}}\ ,\qquad L_b \in [0,1] \tag{6}$$

$$F_{fitness} = e^{-\alpha E}\ ,\qquad F_{fitness} \in [0,1] \tag{7}$$

Where,

m	the maximum number of CFs for evolution process
n	the number of queries
n_{atts}	the number of columns
f	the number of CFs for the current solution
c_i	the needed number of CFs, $i = 1,2, \dots, n$ for i_{th} query.
a_i	the number of columns in CF i, $i = 1,2, \dots f$
s_i	size percentage of CF i, $i = 1,2, .. f$
l_i	loaded frequency of CF i for the query set, $i = 1,2, .. f$

Equation (1) is the cost function. We define this to measure the fitness of each and every solution by applying the weighted sum method [27], which is the best known and simplest multi-criteria decision analysis [28] method for evaluating a number of alternatives in terms of a number of decision criteria. The fitness function is defined in (7). The higher the cost of one solution, the smaller chance it can survive.

In the cost function (1), $\mathrm{C}, \mathrm{S_k}, D, S_c, L_b$ represent the above five defined factors respectively. $\mathrm{W}_1, \mathrm{W}_2, .., W_5$ are the weights of these five factors, and the sum of these weights should be one for normalization. The feature scaling is already integrated as all five factors take values from comparable intervals, which ensures faster convergence. The objective function can be configured by setting the weights for every factor, where the more important factors should have the greater weight value to be set.

3) Operations of Evolution (Recombination)

Operations of an evolutionary algorithm are very important, which conducts the direction of evolution to optimized solutions. An algorithm with good operations can let solutions jump from a local area to search in the global space, which leads the evolution to a better and faster convergence point. For this HBase optimizing problem, genetic operations together with validity-check for evolution process are defined as follows.

a) Mutation

A mutation operator is defined to add or delete one or more columns inside one CF. Redundant columns can be removed to make a CF more effective for query types that don't need those columns, and some other columns can be added so that the CF can match some more query types. To guarantee the validity of the new generated solution, only the column which doesn't exist in the current CF can be added, and only the column which has already appears in other CFs can be removed. Fig. 3 shows a simple example of the mutation operation. While new Individual 3 is created from Individual 1 by adding column a6 in the first CF, new Individual 4 is generated from Individual 2 by deleting column a10 from the first CF.

b) Crossover

The crossover operator exchanges columns between two CFs inside an HBase schema. Such crossover mixes good sub-solutions of CFs without any disruption of the partitions: partial lists of columns, which matches query types, can be well-preserved and recombined between different CFs. Crossover might bring redundant columns from partial parts of two CFs, thus the redundant columns have to be checked and removed to make sure that the generated individual is valid. Fig. 4 shows a simple example of crossover operation. Initially, new Individual 2 is obtained by exchanging columns a1, a2, a3 with a5, a6 and a7 between first and second CF in Individual 1. As the obtained new Individual 2 is invalid due to having a duplicate column a6 in the first CF. It is therefore corrected afterwards by removing duplicate column a6.

c) Split-up and Merge

Considering that HBase schema might have a variable number of CFs, new genetic operations are required for evolution. Split-up operator is designed to split a CF into two smaller CFs inside a solution. A dividing point is chosen by the operator, and the columns before diving point are kept in the current CF. The columns after the dividing point form a new CF. The number of CFs are therefore increased by one after the split-up operation. Merge operator is the reverse to split-up operator, which concatenates two CFs to a single large CF inside one solution. Similar to the crossover operator, duplicate columns need to be checked and removed to make sure that the generated solution is valid. The number of chromosomes is decreased by one after the merge operation. Fig. 5 shows this example of split-up and merge operation. Split-up operation is applied on the second CF in Individual 1 and creates two new CF labeled as c2 and c3 in new Individual 2. Merge operation merges the second and third CF in Individual 3 to form a new CF in new Individual 4, and eliminates duplicate column a9.

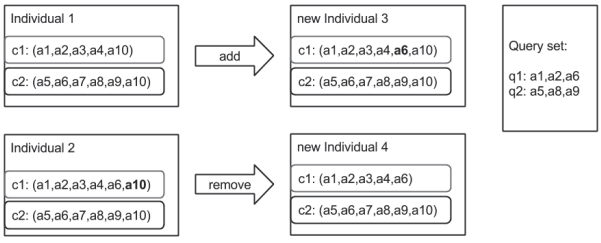

Fig. 3. Example of mutation operation

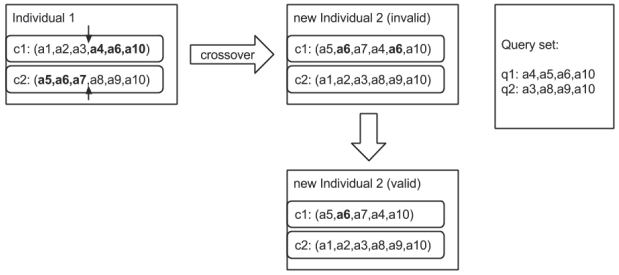

Fig. 4. Example of crossover operation

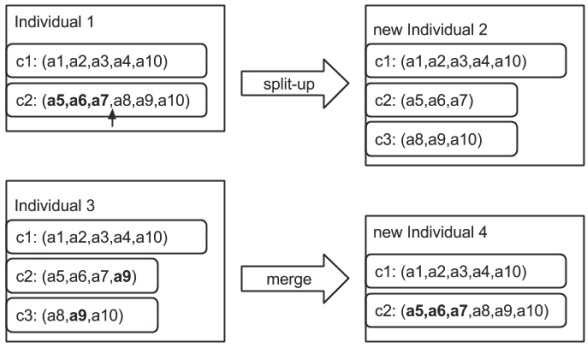

Fig. 5. Example of split-up and merge operation

d) Cross-Fertilization

So far, all defined genetic operators take effect inside one solution, which from the GA perspective is self-evolution. In order to have a stronger operator, which exchanges genetic information across different solutions, the cross-fertilization operator is defined. The cross-fertilization operator acts on two parental solutions. In the operation, CFs will be recombined between two solutions as follows:

1. Choose two solutions as the parent.
2. Take half of CFs as the first half CFs of the offspring from mother. In addition, take half of CFs as the second half chromosomes of the offspring from father.
3. Delete all duplicate columns in the CF from the father that already exist in the CF from the mother.
4. Fill up the missing columns randomly in the chromosomes from father.

Fig. 6 is an example of cross-fertilization. The two parental Individual 1 (mother) and Individual 2 (father) exchange their CFs to create a new Individual 3.

4) Selection Method

In GAs, the Selection Phase aims to keep individuals with good quality, and to eliminate bad individuals. There are many selection methods for evolutionary algorithms. Here we apply proportional selection [29], the most common one. The probability to survive for solution i is calculated as (8)

$$\mathrm{P(i)} = \frac{F_{fitness}(i)}{\sum_m F_{fitness}(m)}, \quad P(i) \in [0,1] \qquad (8)$$

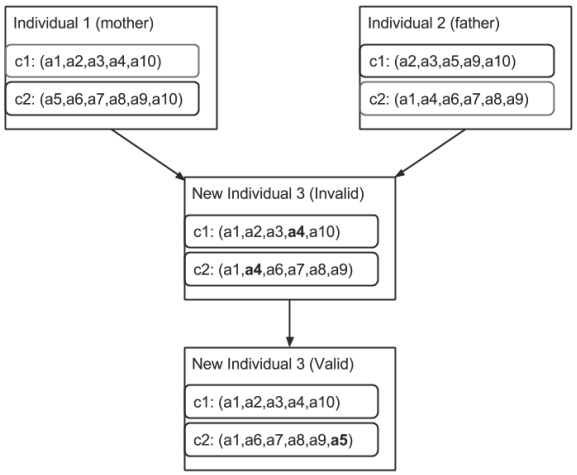

Fig. 6. Example of cross-fertilization operation

V. EVALUATION

The designed EA is evaluated with a real dataset provided by ZANOX AG. The dataset contains 2.6 million rows of aggregated, anonymized tracking data, generated by users seeing and clicking on internet advertising, and 1.3 million historical user queries. The aggregated tracking data will be stored in HBase with the CF schema that should be evaluated by executing user queries that request aggregated statistics of the stored tracking data. The user queries are chronologically divided into parts for training, validation and testing, in order to be able to check the ability of a solution to adapt to trends. While the training and validation queries are used for generating optimized CF schemas, the testing queries are used for the performance tests of the CF schemas.

A. Set up

The evaluation environment is an HBase cluster consisting of five machines. Each machine has one Quad-Core CPU and 16 GB memory. The software setup is Cloudera Standard 4.7.3.

B. Construct Column Family Schemas

In order to test the result of the designed EA, HBase tables with the optimized CF schema **s03** and **s05** found by EA are constructed. Additionally, HBase tables with other reference CF schemas **ref01**, **ref02** and **ref03** are constructed for comparison as in Table 1.

Table 1 Column Family Schemas for Performance Test

Optimized CF Schemas	**s03**	the optimized schemas by the EA with the CF size of three.
	s05	the optimized schemas by the EA with the CF size of five.
Reference CF Schemas	**ref01**	all attributes in one CF.
	ref02	CF schema, which has a high cost value by the defined objective function (1).
	ref03	CF schema, where columns are divided into CFs according to semantics of the stored data.

C. Perfrormance Test

At the beginning of the performance testing, we need to construct HBase tables with these above five CF schemas, and then import the aggregated tracking dataset respectively. After that, the average response time for each HBase table will be measured by the testing query set.

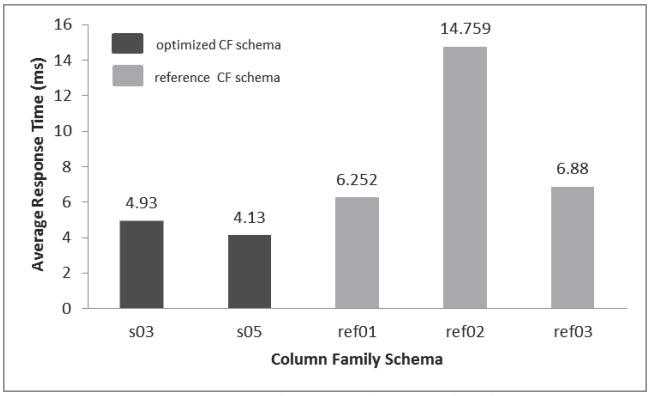

Fig. 7. Average response time of different column family schemas

Fig. 7 shows the average response time for different CF schemas. Among the reference CF schemas, **ref01** has the lowest average response time, while the value of **ref02** is the highest. This result proves a very important phenomenon: no division is better than a bad division for CFs. Consequently, one of best practices while designing HBase schemas in the case where data-access patterns are unknown is to store all columns together [4], although HBase provides a column family schema. However, a well optimized HBase schema can provide a much better reading performance. With three CFs, **s03** can reduce more than 20% response time on average comparing to **ref01**. The average response time can be further minimized if the number of CFs is changed to five. About 34% average response time can be reduced by **s05** comparing to **ref01**, while 72% average response time can be reduced comparing to **ref02**.

D. Result Analysis

As shown in Fig. 7, the optimized HBase schemas **s03** and **s05** got by the designed EA have much lower average response time than others. To prove that this result is not caused by some small parts of test queries by chance, statistical significance [30] check is made. More concretely, paired T-Test [23] is introduced for result analysis.

In statistics, a paired t-test compares two samples in cases where each value in one sample has a natural partner in the other. It measures how different two samples are (the t-value) and tells you how likely it is that such a difference would appear in two samples from the same population (the p-value).

In performed experiments, the measured response time for one CF schema can be regarded as one sample. Two samples for different CF schemas are paired because the response time for different CF schema is measured by the same queries as the sequence of the query dataset.

Table 2 T-Tests for the pairs between optimized and reference CF schemas

T-Test	Pair1 (s03 and ref01)	Pair2 (s03 and ref02)	Pair3 (s03 and ref03)
t-statistic	-131.756	-761.643	-209.583
P-Value (T<t)	P < 0.0001	P < 0.0001	P < 0.0001

Table 2 shows the result of the T-Tests between the optimized CF schema **s03** and other reference CF schemas (**ref01**, **ref02**, **ref03**). As we can see the P-Value of all these three pairs are less than 0.0001, which is far less than the usual significance level 0.01. Therefore, it can be concluded that, the performance improvement for the optimized CF schema has a statistical significance.

VI. Conclusion and Future Work

The paper focuses on designing and developing an EA for the HBase column family schema optimization problem. The proposed EA has a fast convergence and the evaluation results proved with statistical significance that the obtained HBase schema has a better reading performance.

However, the current HBase version does not support dynamic schema change, meaning the system has to be stopped if a new CF schema needs to be applied. It is a challenge for real applications to determine when to update the optimized schema so that the system can get the best performance. But as the open-source HBase system is being actively developed, dynamic schemas may be added as a new feature in future versions.

Besides, the algorithm is evaluated only with the real dataset of the reporting system at ZANOX AG. As the future work, more tests need to be done with different use cases in practice.

Finally, EA is a powerful solution for an optimization problem. In this paper, multi-chromosomes representation and new genetic operators are designed for the HBase column family schema optimization problem. In the future work, more efficient genetic operators and selection methods can be devised for the further optimization of the designed EA. These EA models could also be practical for designing EAs for other optimization problems.

Acknowledgments

Authors gratefully acknowledge the financial support from ZANOX AG, who also granted access to real dataset which made it possible to evaluate HBase schemas with statistical significance tests.

This work is also partially supported by China National Science Foundation (Granted Number 61272438, 61472253), Research Funds of Science and Technology Commission of Shanghai Municipality (Granted Number 14511107702, 12511502704).

The indebted gratitude deserves the anonymous reviewers for their insightful comments, which greatly helped to strengthen the paper. Special thanks also go to Alex Allan, whose suggestions and proofreading helped make the presented matter easier to understand.

References

[1] S. Sumathi and S. Esakkirajan, *Fundamentals of Relational Database Management Systems*, 2007.

[2] X. Li, J. Han, H. Gonzalez, "High-dimensional OLAP: a minimal cubing approach," *30th conf. on Very large databases*, vol. 30, 2004, pp.528-539.

[3] S. Edlich, "NoSql Databases." Internet: nosql-database.org [May. 2014].

[4] L. George, *HBase: The Definitive Guide*, O'Reilly Media, Inc., 2011.

[5] F. A. Y. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, 2008, pp. 1–26.

[6] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, Jun. 2002, pp. 51-59.

[7] A. Aiyer et al., "Storage Infrastructure Behind Facebook Messages: Using HBase at Scale.," *IEEE Data Eng. Bull.*, 2012, pp. 4-13

[8] J. Lin and A. Kolcz, "Large-scale machine learning at twitter," *Proc. of ACM SIGMOD int. conference on Management of Data*, 2012, p. 793.

[9] E. Bortnikov, E. Hillel and A. Sharov, "Reconciling Transactional and Non-Transactional Operations in Distributed Key-Value Stores," *Proc. of Int. Conf. on Systems and Storage*, 2014, pp. 1-10.

[10] B. Sigoure, "OpenTSDB scalable time series database," Internet: opentsdb.net [May. 2014].

[11] M. Doctor, G. Nguyen, "Realtime User Segmentation using Apache HBase Architectural Case Study," *HBaseCon* 2013.

[12] J. Natkins, "Design Patterns for Building 360-degree Views with HBase and Kiji," *HBaseCon* 2014.

[13] A. Jindal, E. Palatinus, V. Pavlov, and J. Dittrich, "A comparison of knives for bread slicing," *Proc. of the VLDB Endowment*, 2013, vol. 6, no. 6, pp. 361-372.

[14] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou, "Vertical partitioning algorithms for database design," *ACM Transactions on Database Systems(TODS)*, vol. 9, no. 4, 1984, pp. 680-710.

[15] M. Castellanos, D. Umeshwar, and R. Miller, *Enabling Real-Time Business Intelligence*, Springer, 2010.

[16] R. A Hankins and J. M Patel, "Data morphing: an adaptive, cacheconscious storage technique," *29th international conference on Very large data bases*, vol. 29, 2003, pp. 417-428.

[17] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, "HYRISE: a main memory hybrid storage engine," *Proc. of the VLDB Endowment*, vol. 4, no. 2, 2010, pp.105-116.

[18] A. Jindal, J. Quiané-Ruiz, and J. Dittrich, "Trojan data layouts:right shoes for a running elephant," *Proc. of the 2nd ACM Symposium onCloud Computing,* ACM, 2011, p. 21.

[19] C. Lam, *Hadoop in Action*, Manning Publications Co., 2010.

[20] J. van Leeuwen, *Handbook of theoretical computer science: algorithms and complexity*, Vol. 1, 1991.

[21] C. W. Ahn, *Advances in Evolutionary Algorithms: Theory, Design and Practice*, 2006.

[22] E. Falkenauer,*Genetic Algorithms and Grouping Problems*,Wiley, 1998.

[23] G.E. Box, J.S. Hunter, and W.G. Hunter, *Statistics for Experimenters: Design, Innovation, and Discovery*, Wiley, 2005.

[24] T. Back, D. Fogel, and Z. Michalewicz, *Handbook of evolutionary computation*, 1997.

[25] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989, p. 432.

[26] C. W. Ahn, "Practical genetic algorithms," *Stud. Comput. Intell*., vol. 18, 2006, pp. 7–22.

[27] E. Triantaphyllou, *Multi-criteria decision making methods: a comparative study*, Springer, 2000, pp. 81–84.

[28] J. Figueira, S. Greco, and M. Ehrgott, *Multiple Criteria Decision Analysis: State of the Art Surveys*, vol. 78, 2005, pp. 859–890.

[29] J. H. Holland, *Adaptation in natural and artificial systems*, May 1992.

[30] F. L. Coolidge and F. L. Coolidge, *Statistics: A Gentle Introduction*, Sage Publications, 2012.