

Data Partition Optimization for Column-Family NoSQL databases

Li-Yung Ho Meng-Ju Hsieh
Research Center for Information
Technology Innovation
Academia Sinica,
Department of Computer Science
and Information Engineering,
National Taiwan University
Taipei, Taiwan
Email: {lyho}@iis.sinica.edu.tw

Jan-Jan Wu
Institute of Information Science,
Research Center for Information
Technology Innovation,
Academia Sinica
Taipei, Taiwan
Email: wuj@iis.sinica.edu.tw

Pangfeng Liu
Department of Computer Science
and Information Engineering,
Graduate Institute of
Networking and Multimedia,
National Taiwan University
Taipei, Taiwan
Email: pangfeng@csie.ntu.edu.tw

Abstract—Data conversion has become an emerging topic in BigData era. To face the challenge of rapid data growth, legacy or existing relational databases have the need to convert into NoSQL column-family database in order to achieve better scalability. The conversion from SQL to NoSQL databases requires combining small, normalized SQL data tables into larger NoSQL data tables; a process called denormalization. A challenging issues in data conversion is how to group the denormalized columns in a large data table into "families" in order to ensure the performance of query processing. In this paper, we propose an efficient heuristic algorithm, GPA (Graph-based Partition Algorithm), to address this problem. We use TPC-C and TPC-H benchmarks to demonstrate that, the column-families produced by GPA is very efficient for large scale data processing.

Index Terms—vertical partition, column partition, column family, NoSQL database

I. INTRODUCTION

Relational model is the most popular approach for data management. In the relational model, data are categorized into tables, and the tables are mutually connected through the table's primary key. A process called *table normalization* maintains only one copy of the data. This relational database structure offers two key benefits. First, with table normalization, there is no redundant data. A user retrieves the information from many tables by joining them through the keys. Second, relational databases provide a user-friendly SQL query interface which has been consistently optimized to improve query processing efficiency. Despite these benefits, the need to frequently join operations between normalized tables in relational databases makes this approach difficult to scale up. As a result, other solutions known as Not-Only SQL (NoSQL) databases have become popular for large scale data management.

This high degree of scalability makes NoSQL databases an important tool for Big Data applications. NoSQL databases are based on the concept of *denormalization*, where the data are duplicated so that one can retrieve data from a single table rather than having to collect them from multiple tables, thus reducing query time and improving scalability. There are

four main categories of NoSQL databases: key-value, column-family, document, and graph. Each type targets a specific kind of data. In this paper, we focus on column-family databases.

The rapid growth in data accumulation and storage in recent years has raised the need to convert existing SQL databases to NoSQL databases (e.g., HBase) to improve database scalability. This requires converting existing relational schemes into NoSQL formats. The main step of this conversion is called *denormalization*.

Although NoSQL databases provide improved scalability, they do not provide sufficient support for SQL queries. Therefore, existing SQL applications or legacy SQL code cannot run on NoSQL systems. To address this issue, researchers and industrial practitioners have developed many SQL-compliant NoSQL systems such as HadoopDB [1], Hive [2] and Trafo-dion [3]. Using such systems, users can submit SQL queries, which are then internally converted into NoSQL-compliant commands for execution. As a result, users do not need to modify their SQL code to take advantage of NoSQL scalability.

In this paper, we study how data conversion affects the query performance on SQL-compliant NoSQL systems. That is, how should we layout the data on the NoSQL system to optimize the processing of the given SQL queries in HBase. HBase [4] is a NoSQL database with a column-family scheme, which implements Google's BigTable [5]. HBase is a write-optimized database supporting multi-tenancy and flexible data schemes with a high degree of scalability, and is a suitable candidate for converting SQL to NoSQL because users can define flexible data schemes as in a relational model. In HBase, a data item is defined as a set of *columns*, which are then grouped into *column-families*. To achieve better performance, we need to consider two important issues.

First, in HBase users can define as many attributes as they want but the number of families should be limited to a small number. Hbase maintains a memory store (memstore) and a physical file (HFile) for each column family [4]. Once the size of the memory store exceeds a certain threshold, Hbase

triggers compaction to flush memstore to an HFile. Because the compaction is region-based, all column families in that RegionServer will perform compaction simultaneously. However, the content for some families may change occasionally, thus frequent compaction will result in unnecessary I/O operations and degrade operational performance. This raises an important question: when we denormalize the data, how do we group the columns into a specified number of families such that the processing cost is minimized with respect to a given set of queries? In this paper, this issue is referred to as the *column partition problem*.

Second, we also consider limiting the number of columns in a family. Because HBase stores the column family data in an HFile, a family containing too many columns results in a very large HFile consisting of many blocks. A large HFile incurs extra overhead if the data access pattern is random and sparse, which is common in transactional queries, because HBase needs to scan a whole block to identify a required key. If a block contains too many columns of unnecessary data, it will result in many unnecessary I/O operations and thus degrade performance. Therefore, our goal is to reduce the number of columns in a column family, and also to group high-affinity attributes in the same column family so that HBase's efficient scanning operation can be used to improve data retrieval performance.

The column partition problem for NoSQL databases is similar to the vertical partition problem for traditional relational databases. They both aim to group high affinity attributes to avoid the loading of unnecessary data. Although the vertical partition problem has been widely studied, most current work on vertical partitioning focuses on determining schemes for minimizing cost partitioning. However, the goal of the column partition problem is to find an optimal partition with a specified number of column families. Therefore, existing vertical partitioning solutions may not be feasible for the column partition problem.

In this paper, we prove that the column partition problem is NP-hard, and propose a heuristic method, called the *Graph-based Partition algorithm* (GPA), to solve this problem for column-family databases. GPA is inspired by the multi-level graph partitioning framework. GPA consists of two phases. The first phase groups columns into the target number of column families, while the second step fine-tunes the results of the first step.

We conducted experiments to evaluate the effectiveness of the proposed algorithm. Results demonstrate that GPA outperforms a state-of-the-art vertical partition method, AVP [6]. The main contributions of this work are as follows.

- We prove that the column partition problem is NP-hard, and we propose a very effective heuristic algorithm, Graph-based Partition algorithm (GPA).
- We compare partition quality from the proposed algorithm with that of an existing vertical partition algorithm. To the best of our knowledge, this is the first effort to study the performance of an existing vertical partition algorithm for a relational database on NoSQL databases.

- We conduct extensive experiments to demonstrate the effectiveness of our algorithm. To ensure the robustness of experimental results, we re-write the query set of TPC-C [7] and TPC-H [8] for HBase.

The remainder of this paper is organized as follows. Section II reviews related work. Section III gives a brief overview of the HBase column-family database. Section IV presents the proposed heuristic algorithm. Section V demonstrates the efficiency of our algorithm and provides a comparison with other approaches, and Section VI provides conclusions.

II. RELATED WORKS

The column partition problem for NoSQL databases is highly related to the vertical partitioning problem for relational databases. In this section, we discuss the several vertical partitioning algorithms and column-store systems. The group of high affinity attributes in vertical partition is called a *fragment*, while in the context of column partition, the group is the *column-family*. We use the term column-family throughout this paper.

A. vertical partitioning algorithm

Vertical partitioning algorithms fall into two categories: (1) *best-fit* vertical partitioning and (2) *N-way* vertical partitioning. The best-fit method generates an overall optimal partitioning that minimizes the processing cost of queries. It has no constraint on the number of column-families it generated. On the other hand, N-way partitioning generates a specified number of column-families required by the users. It also minimizes the processing cost for that number of column-families. In addition to the related work of vertical partitioning, we also discuss existing techniques related to our algorithm.

N-way vertical partitioning is identical to column partition problem because they both have requests on the number of partitions. We will focus on N-way partitioning and discuss some important results of it. Jin Hyun Son and Myoung Ho Kim [6] proposed an adaptable vertical partitioning (AVP) method. AVP can generate not only best-fit vertical partition but also n-way vertical partition in one framework. The authors claim that AVP is the first method to generate two kinds of partition in a single algorithm. In AVP, it generates a *partition tree* from leaves to root (bottom-up). The nodes with the same depth in the tree form a feasible partition configuration. Initially, each leaf node represents a single attribute. AVP chooses two leaf nodes according to the merging profit and generates a parent node of that selected two nodes. The generated parent node contains the two attributes of its children nodes. AVP also generates parent nodes for all unselected leaf nodes. The parent of an unselected node has the same attributes as its child. Because AVP only selects two nodes to merge in each step of the tree generation, it requires $n - 1$ steps to generate the root which contains all n attributes. The authors also develop a cost model to evaluate the data retrieval cost for each partition configuration. Users can either choose a partition with a specified number of column-families or a partition with the lowest cost.

Although AVP can support N-way partitioning, it focuses on traditional databases rather than column-family databases. Moreover, it does not demonstrate its efficiency with real benchmarks. In the AVP paper [6], the authors only show that AVP can achieve less processing cost compared with BVP method based on their proposed cost model. Therefore, existing literatures did not provide information on how AVP performs in real data experiment as well as the performance of AVP for column-family databases.

Zhikum et al. [9] proposed an objective function to mathematically measure the quality of vertical partition when dividing column-family database. The authors claim their function is valid because it can predict the same result as [10]. The authors give a metric to evaluate the partitioning algorithm, however, they did not provide any vertical partitioning algorithm for NoSQL databases.

Curino et al. [11] propose a workload-driven approach for database partition and replication named *Schism*. *Schism* first generates a graph representation according to the workload and tuples in the database. The node in the graph represents the tuple, and the edge weight account for the frequency of co-accessing the two tuple. *Schism* partitions the graph using METIS [12]. According to the partitioning result, *Schism* distributes and replicate the data. Our work is different from *Schism* in two-fold. First, the graph representation is different. We consider the affinity of attributes instead of tuples. Second, *Schism* is interested in distribution and replication of SQL database. In this work, we focus on how to partition the columns of a column-family database.


B. Column-store system

DBDesigner [13] is a customizable physical design tool for columnar database. It partitions the columns into several *projections* and segments the data belong to those projections to the storage nodes according to the user-defined policy and operation mode. Note that a column may belong to several projections. There are two main differences between our work and DBDesigner. First, DBDesigner targets transition relational database, instead, we focus on NoSQL database system so the data processing models are different. Second, DBDesigner does not have constraint on the number of projections, however, in our problem, we need to partition the columns into a specified number of column-families.

C-store [14] is a read-optimized, column-oriented database system. Like DBDesigner, C-store is a relational database and it does not consider how to partition the columns into a specified number of projections. They focus on how to optimize the read operation and the consistency among data.

III. PRELIMINARY

In the relational model, the data is normalized to maintain only one copy of the data. The tables are related by the primary key. Figure 1 shows an example of relational tables. Table *Users* are related to Table *Orders* by the primary key *UserID*. To retrieve the complete information of a record we have to join two tables using the key *UserID*. We can



Orders			
OrderID	UserID	Amount	Time
OD1	U1	1	10:11
OD2	U2	2	11:33
OD3	U2	3	11:59

Users	
UserID	Name
U1	Jack
U2	Eric

Fig. 1. A relational table

Row Key	TimeStamp	Column Family 1		Column Family 2		
		UserID	Name	OrderID	Amount	Time
1	T1	U1	Jack	OD1	1	10:11
2	T2	U2	Eric	OD2	2	11:33
3	T3	U2	Eric	OD3	3	11:59

Fig. 2. A table with column-family scheme

convert the relational tables to a column-family table by denormalization. For example, we transform the tables in Fig. 1 to the table shown in Fig. 2. With denormalization, we duplicate the data "Eric" in the Name column. However, we can obtain the complete information of a record without a JOIN operation as in the table in Fig. 2. This prevents communication among machines in a distributed environment and improves scalability.

HBase is a distributed data store based on a master-slave architecture. An HBase cluster consists of a *master* and a number of *regionserver*s. The master orchestrates the regionserver to manipulate the data. The master also monitors RegionServer health and optimizes data availability by initializing another RegionServer if a RegionServer fails. HBase uses ZooKeeper [15] to coordinate the servers, including master selection, regionserver registration and synchronization. A table in HBase is split into *regions*. A region contains a range of consecutive row keys. A region is the distributed unit and is assigned to a regionserver to manage.

HBase is a column-family database. The main components in this scheme are *columns* and *column-families*. For example, the table in Fig. 2 has seven columns and two families. Column *UserID* and *Name* belong to Column-family 1, and Column *OrderID*, *Amount* and *Time* belong to Column-family 2. Semantically, the columns in the same family have a high degree of affinity and are frequently accessed together. Therefore, HBase stores the data of a family together to allow for efficient access. Column partition becomes even more important for big data processing because it prevents the reading of unnecessary data. For example, suppose we retrieve the data of *UserID*, *Name* and *OrderID* from Fig. 2; since the data of *OrderID* is in Column-family 2, we will need an additional scan for Column-family 2. However, if we store all three columns in the same family, e.g., Column-family 1, the data can be retrieved through a single scan of Column-family 1.

The data model in HBase is a map. We use four identifiers to obtain a value: row key, family name, column name and timestamp. HBase combines these identifiers into a composed key and stores it with the value as a key-value record on HDFS [16]. HBase stores records with the same family name in an *HFile* on HDFS. The HFile consists of several fixed-sized data blocks used to store the records. The HFile also maintains an index of the beginning key for the blocks, such that HBase can efficiently find the corresponding block containing the target record. Although the HFile provides the index for HBase to quickly identify a block, the HBase scanner needs to perform a full block scan to retrieve the required data.

IV. ALGORITHM

In this section, we formally define the column partition problem, prove that the problem is NP-hard, and then propose an effective heuristic algorithm to solve it.

A. Column Partition

Given a set of attributes $A = \{a_1, a_2, \dots, a_m\}$, a *column partition* $P = \{p_1, p_2, \dots, p_k\}$ of A divides A into k disjointed sets. Column family p_i is a subset of A . The number of columns in p_i is denoted as $|p_i|$. We limit the number of columns in a family by a bound B , that is, $|p_i| < B \forall i$. A feasible partition P of A satisfies the following conditions.

- $p_i \subset A$ and $|p_i| < B \forall i \in 1, \dots, k$
- $A = \cup p_i$
- $p_i \cap p_j = \emptyset$ for $i \neq j$

This raises an interesting question: What is the partition to minimize the overall query processing cost? We first define the cost before answering this question.

B. Cost model

Suppose we have a set of queries $Q = \{q_1, q_2, \dots, q_n\}$, and RA_{q_i} is a subset of A required by query q_i . The *processing cost* of a query q_i is defined as the total number of columns in the retrieved column families. For example, suppose we have two column families X and Y . X contains two columns a_1 and a_2 and Y contains three columns a_3 , a_4 and a_5 . Let $RA_{q_1} = \{a_1, a_2, a_3\}$. Since q_1 needs to retrieve X and Y to collect the necessary data, the retrieval cost θ_{q_1} of q_1 will be $|X| + |Y| = 2 + 3 = 5$. HBase stores the data of a column-family in a single HFile. To retrieve the data in a column-family, the scanner first queries the index section of the HFile to identify the corresponding blocks. For each identified block, the scanner scans the whole block to retrieve the data. The range of a row key in a block is proportional to the number of columns in that family. Therefore, in the worst case, the overall searching cost is proportional to the total number of columns for the retrieved families. Note that we assume the data size of the columns is the same, so the cost is simplified to the number of columns in the column-family.

We now formally define the cost θ_{q_i} of query q_i . Let $CF(a_i)$ denote the column for the family containing attribute a_i . The set Φ_{q_i} denotes the set of the column family required by query q_i . That is

$$\Phi_{q_i} = \cup_{a_j \in RA_{q_i}} CF(a_j)$$

The cost θ_{q_i} of query q_i is defined as

$$\theta_{q_i} = \sum_{cf \in \Phi_{q_i}} |cf|$$

where $|cf|$ denotes the number of columns in the column family cf .

With the cost definition, the decision version of the column partition problem is to determine whether there is a feasible partition such that the overall processing cost is bound by κ .

Definition 1: Given a query set Q , attribute set A , RA_{q_i} for $i \in \{1, \dots, n\}$ and positive numbers k , B and κ , we want to find a partition P of A , where

- $P = \{p_1, p_2, \dots, p_k\}, p_i \subset A, |p_i| \leq B \forall i$
 - $\cup p_i = A$
 - $p_i \cap p_j = \emptyset$ for $i \neq j$
- and the overall cost $\sum_{i \in \{1..n\}} \theta_{q_i} < \kappa$.

C. NP-Hardness

We prove that the column partition problem is NP-hard by reducing the graph partition problem (GPP) [12] to it. In GPP, given a graph $G = (V, E)$ and positive numbers k , B and λ , we want to find a partition P of V , such that

- $P = \{p_1, \dots, p_k\}, p_i \subset V, |p_i| < B \forall i \in 1, \dots, k$
- $p_i \cap p_j = \emptyset$ for $i \neq j$
- $\cup p_i = A$

and the number of *cut-edges* is less than λ . An edge is a cut-edge if its two endpoints belong to different partitions. Note that GPP is also NP-hard if we set $k = 2$ and $B = \frac{|V|}{2}$. That is, we partition the graph into two even parts. In the following proof, we set $k = 2$ and $B = \frac{|V|}{2}$ for both CPP and GPP.

Theorem 1: The column partition problem is NP-hard.

Proof. We reduce GPP to CPP. The details of the proof is omitted due to space constraint. ■

D. Graph-based Partition algorithm

We propose a heuristic algorithm to address CPP due to its NP-hardness. We name it the *Graph-based Partition algorithm (GPA)*. GPA is inspired by the multi-level graph partitioning algorithm [12]. GPA is designed based on the concepts of *graph partitioning* and *refinement*. Multi-level graph partitioning is a framework used to partition a graph. The goal is to partition a graph into a specified number of balanced parts while minimizing the number of cut-edges. In this framework, a graph is coarsened into a series of smaller graphs to reduce the problem size. Once the graph is small enough, it is partitioned to a specified number of parts. The partitioned graph is then uncoarsened to a series of larger graphs until the original graph is obtained. In the process of uncoarsening, we refine the partition by exchanging the vertices between the two parts to reduce the number of cut-edges.

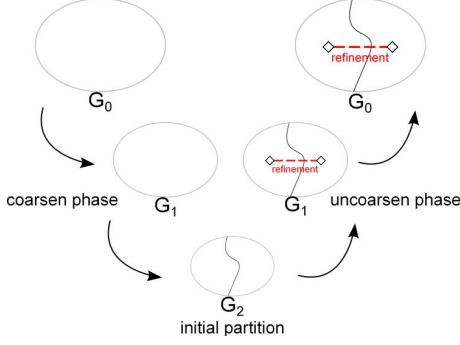


Fig. 3. Multilevel partition framework

Figure 3 shows an overview of the multi-level framework. The original graph G_0 is coarsened to a smaller graph G_1 , which is then further coarsened to G_2 . The size of G_2 is below the threshold so we perform the initial partition to split the graph (into two parts in this example). In the uncoarsening phase, we uncoarsen the vertices to obtain a larger graph. Furthermore, the framework refines the partition by exchanging the vertices near the boundary to reduce the number of cut-edges (indicated by the dashed lines).

GPA consists of two phases - *graph partition* and *refinement*. The graph partitioning phase generates an *affinity graph* based on the queries. We then partition this graph into the required number of partitions using METIS [12]. With the initial partition, the refinement phase refines the partition by exchanging the vertices between families to obtain a partition at reduced cost. The idea is similar to the Kernighan-Lin refinement algorithm [17] in GPP. We exchange the vertices near the boundary of column-families to see if we can reduce costs.

In the refinement phase of the multi-level framework, the movement of a coarsened vertex represents the movement of a group of vertices. Since we do not have a coarsening phase analogous to that in the multi-level framework, each vertex in our graph is just a single vertex. As a result, we can only pick a single vertex and exchange it with another column-family. However, this single vertex movement may not be efficient if a vertex is located in a highly connected group of vertices. The best policy is likely to move the vertices as a group rather than individually. Moreover, movement as a group can retain the graph's topological structure and achieve better data locality. Therefore, we have to consider the movement of a group of vertices in addition to single vertex movement. To define the group of vertices for movement, we first define the *boundary set* for each column-family by performing a limited-depth BFS on each boundary vertex of that column-family. If we want to move a group of n vertices, we first select two column-families and find all possible n -combinations of vertices from the two corresponding boundary sets. We then exchange each pair to see if we can obtain a lower cost.

1) *Graph partition*: The graph partition phase generates an affinity graph according to a given query set. Each attribute

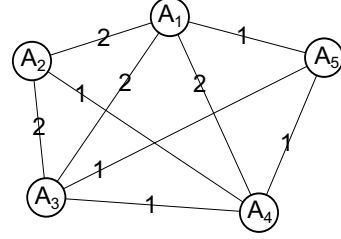


Fig. 4. Affinity graph

represents a vertex in the affinity graph. An edge in the affinity graph means there are queries that simultaneously access both end-point attributes. The weight on that edge represents the frequency of such simultaneous access. For example, Table I shows a set of input queries. A cell t_{ij} in this table indicates whether query i accesses attribute j . "1" means query i accesses attribute j and "0" otherwise. We then generate the affinity graph as shown in Fig. 4. The number on the edge is the edge weight. The weight of the edge between A_1 and A_2 is 2 because A_1 and A_2 are accessed together in two queries Q_2 and Q_4 . We partition the affinity graph into k column-families using METIS.

	A_1	A_2	A_3	A_4	A_5
Q_1	1	0	0	1	1
Q_2	1	1	1	0	0
Q_3	0	0	1	0	1
Q_4	1	1	1	1	0

TABLE I
THE INPUT QUERY SET

2) *Refinement*: With k column-families of the affinity graph, we refine the partition by exchanging the boundary vertices of column-families to reduce the cost. We define B_f^j which represents the set of vertices visited in the breadth-first search rooted at vertex i with depth j . A *boundary vertex* v_b of column-family f is a vertex of f and one of its neighbors is located at another column-family g , $f \neq g$. The set D_f is the union of all boundary vertices of column-family f , that is $D_f = \cup v_b$. We define the *boundary set* S_f^l of column-family f with depth l as follows

$$S_f^l = \cup_{v_b \in D_f} B_l^{v_b} \cap f$$

For any pair of column-families, we can exchange a group of vertices in their boundary sets. The number of vertices in the group is increased from one to a specified number β .

V. EXPERIMENTS

In this section, we describe experiments conducted to demonstrate GPA efficiency and compare the performance of GPA, AVP [6] and a random partition approach. The experiments were performed on HBase with TPC-H and TPC-C query benchmarks and measured the execution time of queries with different column partitions produced by the three algorithms. We also examine the scalability of different partitions.

A. Query scenario

We assume users submit SQL queries to a SQL-compliant NoSQL system using HBase as the backend. To emulate the queries, we take TPC-H and TPC-C to be the benchmark suites. TPC-H focuses on decision support benchmarks, which consists of a suite of business-oriented ad-hoc queries. On the other hand, TPC-C is a suite of on-line transaction processing benchmarks. We use these two benchmark sets to emulate the diversity of user SQL queries.

B. Query transformation

To support TPC-H and TPC-C queries on HBase, users have to transform the queries into a series of processes. HBase only supports preliminary data manipulation interfaces like **Get** and **Set** and it does not support predicates like **WHERE**, **JOIN** and **OR**. As a result, users have to either implement the query logic in the coprocessor function [5] and execute it on each regionserver, or gather all results and process them in the client application. In this experiment, we implement the queries in the coprocessor function because it is more efficient to process in parallel on regionserver. We focus on the coprocessors' performance of the queries, that is, the time spent by the coprocessor to retrieve or modify the data.

We re-write TPC-H and TPC-C queries to different HBase query scenarios due to their different characteristics. TPC-H queries examine large volumes of data and have a complex execution plan. Therefore, the transformation of TPC-H queries is accomplished using an HBase scanner to efficiently search and filter data. On the other hand, in most cases TPC-C queries only require a small set of data, so we leverage the **Get** interface to facilitate small-set data searching.

We decompose a query into three steps in HBase. First, we identify the columns used in the query, and add these columns to the scanner. We then initialize a filter according to the query requirements. In the second step, HBase runs the scanner or **Get** to retrieve data, and the data is processed or update by the coprocessor function according to the query's manipulation logic. In the third step, the results of each regionserver are returned to the client for further processing.

For example, suppose we have a TPC-H query as follows.

```
SELECT  min(c1)
FROM    table
WHERE   c2=c3
AND     c4 = 1
```

The **min** function selects the record having a minimum **c1** value from the retrieved data. The retrieved data must meet the condition where its values of **c2** and **c3** are equal and the value of **c4** must be equal to 1. To transform this query through the abovementioned three steps, we first add **c1**, **c2**, **c3** and **c4** to the scanner and setup the value filter for **c4**. In the second step, HBase runs the scanner to retrieve the records for which the value of column **c4** equals 1. It also returns data including the rowkey and values of columns **c1**, **c2** and **c3**. We then execute the coprocessor function on each regionserver to find all records having the same value for columns **c2** and **c3**.

Finally, the records found on the regionserver are returned to the client to calculate the minimum value of column **c1**.

We are most interested in the time used by coprocessor, which is defined as *data processing time*. We seek to determine the efficiency of a scanner (**Get**) and **UPDATE** in HBase with different column partition configurations. We are not concerned with the time consumed in Step three because the data is processed on the client side and thus has no bearing on how the columns are partitioned. Note that we do not use a compiler to automatically transform the queries, so we do not count the time used for Step one. Therefore, we use a query's data processing time as our performance metric. We execute each query ten times and calculate the average for various column partition configurations.

C. Algorithm Comparison

Experimental results are compared for the AVP algorithm [6], a random partition approach and the proposed GPA algorithm. AVP generates a minimized N-way column partition according to its cost model and we pick the partition with three column-families.

The input for AVP is a 0-1 matrix. The element of the i_{th} row and j_{th} column in the matrix indicates whether the i_{th} query uses j_{th} column (1) or not (0). AVP then generates a partition tree and we pick the partition with three column-families. We also compare our algorithm with a *RANDOM* approach, which means we randomly partition the columns into a specified number of column-families to obtain the column partition. We generate three random partitions and measure the query performance for each. We then take the average performance of the three partitions and compare it with AVP and GPA.

D. Setting

The environment is a 10-node cluster. Each machine is equipped with 12G memory and two Xeon E5-2620 2GHz CPUs. Each CPU has six cores with hyper-threading. The machines are connected by an ethernet gigabit switch. We use HBase 0.94 as our column-family database. The benchmarks are the TPC-H and TPC-C query sets, for which synthetic data are generated. The table contains one millions records, and TPC-H and TPC-C respectively have 62 and 91 columns. Throughout the experiments, the number of column families is set to three based on HBase documentation [4].

E. Results

1) *TPC-H*: Figure 5 compares the data processing time for the three methods, with GPA providing an average 2.25x and 3.09x speedup over AVP and RANDOM, respectively. AVP performance is hampered by its highly imbalanced partitioning. The largest family AVP generates for TPC-H contains more than 50 columns, and the corresponding HFile stores all the data of these 50 columns in the same row key. Full block scanning thus entails considerable waste. On the other hand, although RANDOM has balanced partitioning, the randomly distributed columns make the query inefficient because a query

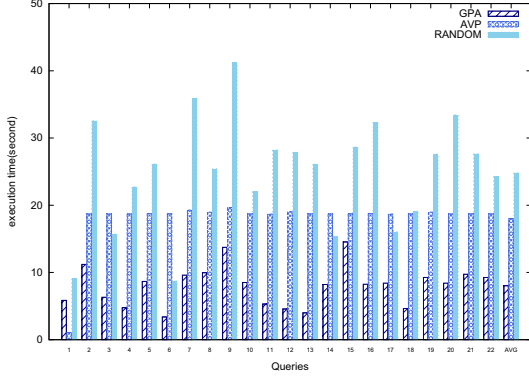


Fig. 5. TPC-H performance for different partition strategy

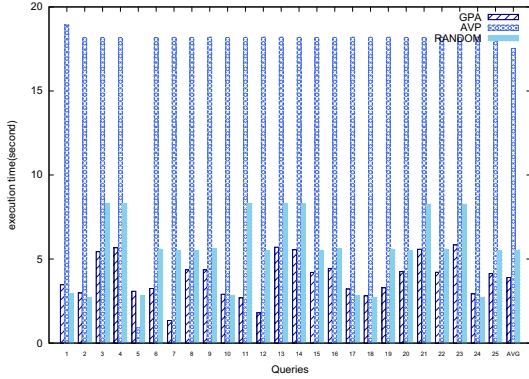


Fig. 6. TPC-C performance for different partition strategy

has to open three HFiles to gather the required data which induce significant I/O overhead. GPA collects commonly-accessed data in a family (and HFile) such that the scan is more efficient. Note that in Query 1, AVP significantly outperforms GPA and RANDOM. This is because AVP groups the required columns of Query 1 in a very small column family, which only contains two columns. This is very efficient in retrieving all the data from a small column family.

2) *TPC-C*: Figure 6 shows dramatic improvements in the TPC-C benchmark. On average, GPA produces a 4.42x and 1.40x speedup over AVP and RANDOM respectively. Again, the largest family generated by AVP contains more than 80 columns, and the resulting unnecessary scanning significantly degrades system performance, especially for the queries which only require the processing of small data sets. Note that the time required for Query 5 by GPA, AVP and RANDOM is 3.08, 0.91 and 2.83 seconds respectively. AVP has a fast response time because it produces a small family containing all the columns required by Query 5. The small family only contains 5 columns, 4 of which are used in Query 5. As a result, the scanning is very efficient.

RANDOM significantly outperforms AVP and performs similarly to GPA. This is because in TPC-C, most of the queries only involve a few columns in a single row. For example, the query updating the items in a customer's shopping

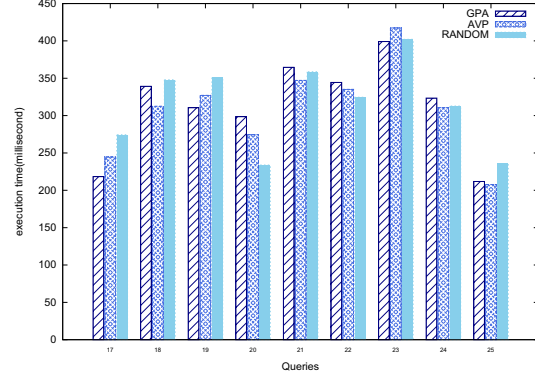


Fig. 7. TPC-C write latency for different partition strategy

cart only involves two columns (one for the shopping cart and the other for the customer ID). In this situation, balancing the column family is more important than grouping the affinity columns because most columns are not so closely related. GPA and RANDOM have a balanced column partition, so they have similar performance.

We further investigate the write latency of TPC-C benchmark. There are nine queries (Query 17 - 25) in TPC-C containing the UPDATE or INSERT predicates. For these nine queries, we measure the read and write latency separately. We take Query 17 for example. Query 17 updates the value of column `w_ytd` if the value of column `w_id` equals a specified number.

```
UPDATE warehouse SET w_ytd = w_ytd + 734
WHERE w_id = 1"
```

The coprocessors first determine the value of `w_ytd` if its `w_id` equals to 1 (read latency), and then update the value of `w_ytd` by adding 734 to it (write latency). Note that HBase is a multi-version database, either UPDATE or INSERT creates a new record with latest version in HBase.

As shown by Fig. 7, among the three compared algorithms, the write latency of Query 17-25 has no significant difference because HBase caches the updates in the memory. For the write operation, HBase first writes the updates to the memory table (memtable). Each column family has its own memory table. Once the total size of all memory tables exceeds a given threshold, HBase performs compaction to flush all the memory tables to disk files. Note that one memory table is flushed to one disk file. Because the updates are cached in the memory table, what kind of column partition strategy does not affect the write latency.

3) *Scalability*: The GPA performance improvement over AVP and RANDOM is also examined using data sets of various sizes, with the number of records in the tables ranging from 500,000 to 8 million. As seen in Fig. 8, for TPC-H query TPC-H performance improves as the number of records grows because larger data sets put a premium on avoiding reading unnecessary data while maintaining data locality. AVP achieves data locality by grouping most of the columns in

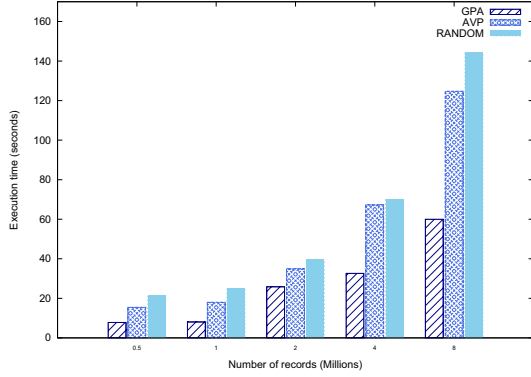


Fig. 8. TPC-H scalability

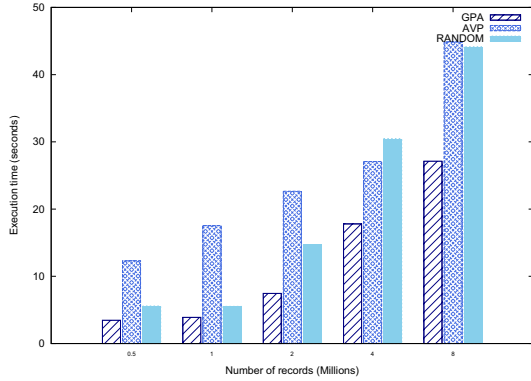


Fig. 9. TPC-C scalability

one column family, however this results in unnecessary data being read. On the other hand, RANDOM avoids reading unnecessary data by evenly distributing the columns into families. Nevertheless, the high affinity columns do not group in the same column family and data locality is lost. GPA achieves both criteria by grouping high affinity columns into balanced column families.

On the other hand, as shown in Fig. 9, TPC-C performance improvement is reduced as the number of records grows. This is because `Get` suffers more overhead when we retrieve data from several families. Since AVP generates a very large family, it is likely to obtain data from the same row in the same family. On the contrary, GPA needs to touch additional families to compile a complete data set for a given record. In the case of TPC-H, GPA provides more stable performance because using the scanner to retrieve data is more efficient than using `Get` for large data sets. Although the speedup is reduced as the number of records grows, GPA still provides a 1.65x speedup over AVP and RANDOM for the largest data set.

VI. CONCLUSION

In this paper, we consider the column partition problem for the transformation from relational to column-family database schemes. We prove that the column partition problem is NP-

hard and propose an efficient heuristic algorithm, the Graph-based partition algorithm (GPA), to address it.

GPA is inspired by the multilevel graph partition framework, and features two techniques to minimize partitioning cost. The first is the *Graph partition* phase, which minimizes the cost while balancing the partition. The second is the *refinement* phase, which further minimizes the cost by exchanging columns between column-families.

Extensive experiments were conducted to evaluate GPA efficiency. We re-write TPC-C and TPC-H queries to obtain transactional and decision support queries for HBase as a basis for comparison. Experimental results demonstrate that GPA outperforms an existing state-of-the-art vertical partitioning algorithm, AVP, providing a 2.25x and 4.42 speedup for TPC-H and TPC-C, respectively.

REFERENCES

- [1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 922–933, Aug. 2009.
- [2] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009.
- [3] "Trafodion," https://wiki.trafodion.org/wiki/index.php/Main_Page.
- [4] "Hbase," http://hbase.apache.org/apache_hbase_reference_guide.pdf.
- [5] F. 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 Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.
- [6] J. H. Son and M. H. Kim, "An adaptable vertical partitioning method in distributed systems," *Journal of Systems and Software*, vol. 73, no. 3, pp. 551–561, 2004.
- [7] "Tpc-c," http://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-C_V5-11.pdf.
- [8] "Tpc-h," http://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpch2.17.1.pdf.
- [9] Z. Chen, S. Yang, H. Zhao, and H. Yin, "An objective function for dividing class family in nosql database," in *Computer Science & Service System (CSSS), 2012 International Conference on*. IEEE, 2012, pp. 2091–2094.
- [10] S. Chakravarthy, J. Muthuraj, R. Varadarajan, and S. B. Navathe, "An objective function for vertically partitioning relations in distributed databases and its analysis," *Distributed and parallel databases*, vol. 2, no. 2, pp. 183–207, 1994.
- [11] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 48–57, 2010.
- [12] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.
- [13] R. Varadarajan, V. Bharathan, A. Cary, J. Dave, and S. Bodagala, "Dbdesigner: A customizable physical design tool for vertica analytic database," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, March 2014, pp. 1084–1095.
- [14] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A column-oriented dbms," in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05. VLDB Endowment, 2005, pp. 553–564.
- [15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.
- [16] "Hdfs," <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>.
- [17] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell Systems Technical Journal*, vol. 49, no. 2, 1970.