# Heterogeneous Replicas for Multi-dimensional Data Management

Jialin Qiao[1,2], Yuyuan Kang[1,2], Xiangdong Huang[1,2(✉)], Lei Rui[1,2], Tian Jiang[1,2], Jianmin Wang[1,2], and Philip S. Yu[3]

[1] KLiss, MOE; BNRist; School of Software, Tsinghua University, Beijing, China
{qjl16,kyy19,rl18,jiangtia18}@mails.tsinghua.edu.cn,
{huangxdong,jimwang}@tsinghua.edu.cn
[2] Research Center for Big Data, Tsinghua University, Beijing, China
[3] University of Illinois, Champaign, IL, USA
psyu@uic.edu

**Abstract.** Multi-dimensional data is widely used in different scenarios, such as cluster monitoring and user behavior analysis for web services. The data is usually managed by distributed databases with a replication strategy, which enhances the availability, fault-tolerance, and I/O throughput. Normally, these replicas share the same physical layout on the disk, which is designed by database administrators according to the target workload. However, it is critical to derive an optimal layout that benefits as many queries as possible, because a layout that accommodates only some queries can negatively impact the others. To tackle this limitation, we propose heterogeneous replicas for multi-dimensional data that provide a higher query throughput without additional disk occupation and without slowing down the writing speed, while still ensuring high availability and load balance. The proposed replication method allows different replicas to be logically identical while having different physical data layouts on the disk. We verified the efficiency of our method in a NoSQL system, Cassandra, with the TPC-H dataset and with a synthetically generated dataset. The results show that our method outperforms state-of-the-art solutions.

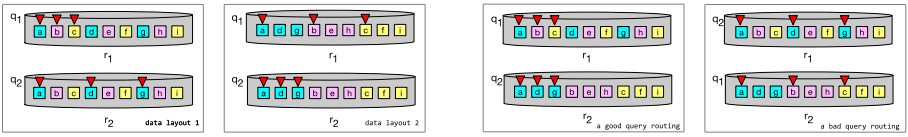
**Keywords:** Multi-dimensional · Heterogeneous · Replica · Layout

## 1 Introduction

With the development of big data technologies, multi-dimensional data is becoming increasingly popular. For meteorological monitoring, for example, a meteorological station collects metrics with various dimensions, such as temperature, rainfall, wind direction, and timestamps for weather forecasts [21]. The data in all of these has multiple dimensions that are sortable and allow for filtering.

To support fast queries, the physical layout of data on a disk plays an important role, especially for multi-dimensional data management. This is because the


Y. Nah et al. (Eds.): DASFAA 2020, LNCS 12112, pp. 20–36, 2020.
https://doi.org/10.1007/978-3-030-59410-7_2




(a) Layouts of uniform replicas

(b) Two routing strategies of heterogeneous replicas

**Fig. 1.** Uniform replicas and heterogeneous replicas (Color figure online)

data can be sorted in multiple dimensions, and the order of these dimensions will ultimately impact the number of I/O operations, which can be time consuming. Therefore, one duty of database administrators (DBAs) is to find the best schema (and therefore the best physical data layout on the disk) for the data according to the query workload. Indeed, it is critical to find an optimal layout that benefits all queries, because a layout that accommodates only some queries can negatively impact the others.

Although distributed storage systems that use a replication strategy are widely used, DBAs benefit little from replication strategies when designing the schema: replicas behave as slaves, and a read operation can be routed to any one of them such that the query load is spread across the nodes. However, if a query runs slowly on one node because of an unsuitable physical layout, rather than the overhead of the node, routing the query to other nodes is of no use. This is because the physical layout of the data on the disk on all nodes is the same; that is, the replicas are uniform.

For example, Fig. 1(a) shows two kinds of data layouts. In the figure, $r_1$ and $r_2$ are two replicas of a dataset, and each data item has two attributes: the character ($a$–$i$) and the color (blue, red and yellow). There are two queries on the dataset: $q_1$ selects the data whose character precedes "$d$" in alphabetical order, and $q_2$ selects the data whose color is "blue". The results are marked by the red triangle in the figure. In data layout 1, the data from replicas $r_1$ and $r_2$ is serialized on the disk by the order of the characters. The cost of $q_1$ is 4 (by scanning from "a" until "d") and the cost of $q_2$ is 9 (by scanning from "a" until "i"), regardless of which replica serves the two queries. In data layout 2, by contrast, the data from the two replicas is serialized by the order of color. Inversely, however, the cost of $q_2$ is 4 and the cost of $q_1$ is 9. That is, uniform replicas cannot benefit all queries, regardless of the data layout on the disk.

Because the "*one size fits all*" model fails to take full advantage of replicas, we propose a new replication mechanism for multi-dimensional data, called **heterogeneous replicas**, to maximize query throughput (i.e., the number of queries completed in a period of time). With our method, replicas are logically identical, despite having different physical layouts. As such, different replicas can be customized for different queries to make full use of the replicas for queries. Figure 1(b) shows an example of heterogeneous replicas: $r_1$ serializes data in alphabetical order, and $r_2$ serializes the same data clustered by color. In the left

part of the figure, if we route $q_1$ to $r_1$ and $q_2$ to $r_2$, then both $q_1$ and $q_2$ have the same cost (i.e., the cost is 4). In this way, the system benefits more from heterogeneous replicas than from the traditional replication strategy. Moreover, routing queries to suitable replicas is important. In the right part of Fig. 1(b), if we route $q_1$ to $r_2$ and $q_2$ to $r_1$, then both $q_1$ and $q_2$ have a cost of 9, degenerating to uniform replicas.

The above example shows how heterogeneous replicas can maximize the query throughput. To use the mechanism, two questions must be answered: "how do we design the data layout for each replica?", and "how do we route queries to suitable replicas?".

Current methods of accelerating queries inevitably bring additional costs. They either optimize limited kinds of queries by adjusting the layout of data, or they optimize more kinds of queries by reduplicating data or building indices at the cost of disk space and write performance. For example, NoSE [11] designs the best schema of column families in NoSQL systems according to a given conceptual model and query workload. NoSE can be seen as offering an optimal uniform replica layout. Considering the diversity of replicas, DivgDesign [4] was proposed to generate different replica configurations. It treats the underlying database as a black box and can be used in our scenario. We compared these two methods to our proposed method in the experiments. Trojan [7] and diversified cache [22] focus on replica design, respectively adopting a Hadoop distributed file system (HDFS) and a distributed cache system. However, these methods cannot be applied to multi-dimensional data management. Another work [18] generates different secondary indices for different replicas such that they are specialized for a specific subset of the workload. However, this incurs overhead on the write speed and disk occupation.

Unlike these approaches, our method of constructing heterogeneous replicas is designed to maximize the query throughput of multi-dimensional data management systems. It can be applied to databases that support multi-dimensional data management, and it avoids additional disk occupation more effectively than the traditional replication strategy. We summarize our contributions in this paper as follows:

- We define and formulate the heterogeneous replica construction problem for multi-dimensional data to achieve the best query throughput for a given workload.
- We propose a new heterogeneous replica construction algorithm and an efficient routing strategy to derive a near-optimal layout that contains different replicas.
- We verified our method on a NoSQL system and conducted extensive experiments that show that our method outperforms state-of-the-art solutions.

The remainder of this paper is organized as follows. We introduce related work in Sect. 2. The workload and problem definition are introduced in Sect. 3. In Section 4, we present a cost model and an efficient routing strategy. The experimental evaluation is reported in Sect. 5. Finally, we conclude the paper in Sect. 6.

## 2 Related Work

Partition Attributes Across (PAX) is a typical method of adjusting the data layout to improve queries [1]. It stores different attributes in a columnar format inside each page on the disk to utilize the CPU cache fully. A column ordering strategy [2] was proposed for large-scale log data stored in Parquet [12] on HDFS [3]. By adjusting the column order stored on the disk based on the query access pattern, it reduces the overall seek cost when accessing multiple columns to accelerate queries. In the work of Rabl et al. [13], data is partitioned by different granularities according to the known workload. An optimal replication factor (i.e., the number of replicas) is selected for each partition, and a partition allocation strategy maximizes system throughput. HYRISE DBMS automatically partitions tables into vertical partitions of varying widths depending on how the columns of a table are accessed by queries [6]. NoSE [11] was proposed to guide and support the schema design of Cassandra. Given a conceptual model of the data required by the application and the workload, NoSE recommends the best schema and query plans based on it. These approaches optimize data layout for queries without considering replicas. Unlike these approaches, we focus on constructing a heterogeneous replica inside each partition. Therefore, data partitioning methods are orthogonal to our work, and can work together with heterogeneous replicas.

DB2 Advisor [19] is an index recommender for IBM's DB2 universal database. Given a query workload and the statistics of the database, it recommends indices by modeling the index selection problem as a variant of the Knapsack Problem. RITA is an index advisor for fully replicated databases [18]. Multiple indices are selected in different replicas. Improvements made by generating materialized views or selecting indices come at the cost of a large extra space budget in addition to the basic data size. Furthermore, maintaining indices and the materialized view can slow down the insertion speed. For example, Cassandra provides a limited form of secondary indexing, and many applications do not use this option for performance reasons [11]. Heterogeneous replicas can be thought of as a restricted form of materialized views [16] in that they are the only data layout, rather than auxiliary structures [10]. Classical materialized views also contain aggregation, joins, and other query constructs that do not exist in replicas.

Fractured mirrors [14] is a method that generalizes RAID 1 to use a hybrid DBMS architecture, where it keeps both the N-ary Storage Model and the Decomposition Storage Model [5] inside each mirror. To the best of our knowledge, this is the first work that applies a layout with different replicas, although it is limited to scenarios that contain two replicas. Trojan generates a layout for each replica in HDFS to optimize a subset of the workload [7]. It benefits from grouping the frequently accessed columns together in each replica, a process known as vertical partitioning [15]. Distorted replicas [8] is a method of restructuring replicas for document-stores. However, these works are specialized to specific data models, which lose applicability in the multi-dimensional data. C-Store [17] and its commercial database, Vertica [10], leverage projections (column groups) to avoid overhead from record reconstruction. However,

it is unclear how the number of projections is determined and how projections are generated—the main topics of our study. Divergent design [4] for leveraging replication was proposed to tune databases more effectively. It works much like the $k$-means clustering algorithm. We compare our proposal with this approach in Sect. 5.

## 3 Problem Statement

We first introduce the data model and query workload. Then, we define the problem and prove its hardness.

### 3.1 Data Model

**Multi-dimensional Data:** Multi-dimensional data refers to a dataset with multiple sortable dimensions representing the attributes of the data. Each record in the dataset al.so contains columns for metrics. As multi-dimensional data is usually organized on the disk according to the order of the dimensions, rather than metrics, we omit the metrics. We use $P = \{d_1, d_2, ..., d_n\}$ to represent the multi-dimensional data model, in which $d_i$ is a dimension. Data is sorted by $d_1$, followed by $d_2$, etc. Given a **data item**, $p_j \in P$, $p_j$ can be formalized as

$$p_j = (p_j.d_1, p_j.d_2, ..., p_j.d_n)$$

where $p_j.d_i$ is the value of $p_j$ in dimension $d_i$. Given two data items $p_1$ and $p_2$, the values in each dimension are comparable. Given a data model with $n$ dimensions, $\{d_1, d_2, ..., d_n\}$, $A = \{k_1, k_2, ..., k_n\}$ is a permutation of the set of dimensions, i.e., $\forall i \in [1, n], \exists j, d_i = k_j$.

**Workload:** The target workload is described as a set of query and insert operations. We focus here on queries, and we defer discussion of insertions to Sect. 5.5. We use $Q$ to represent the known query workload, which is defined as a sequence of query instances. Each query instance $q$ consists of arbitrary predicates connected with and/or. We transform the predicate into a disjunctive normal form that contains $x$ conjunction normal forms (CNFs). The predicate $\sigma(d_i)$ on each dimension is either a range predicate ($d_i \in [s_i, e_i)$) or an equality predicate ($d_i = v_i$). Dimensions in a CNF that do not have a predicate are seen having a range predicate with 100% selectivity. Therefore, a query $q$ is

$$q = \{CNF_1 \vee CNF_2, ..., \vee CNF_x\}, CNF = \{\sigma(d_1) \wedge \sigma(d_2), ..., \wedge \sigma(d_n)\}$$

This query model captures the functionality that is commonly present in multi-dimensional stores.

### 3.2 Problem Definition

Following the intuition that a query has different latencies when executed on replicas with different layouts, our goal is to find an optimal layout for all replicas to maximize query throughput, called the **M**ulti-dimensional data **R**eplica **C**onstruction **P**roblem (MRCP). We introduce the procedure in the following section and provide a formal definition of the MRCP.

Let $R = \{r_1, r_2, ..., r_N\}$ be the $N$ replicas, each with a specific data layout on the disk. Given a workload $Q$, the query router sends each query instance $q$ to a specific replica. Then, each replica (e.g., $r_i$) is assigned with a subset workload (e.g., $Q_i$). We use $Cost(q, r)$ to represent the cost of query instance $q$ on replica $r$. The total overhead for $Q_i$ on $r_i$ is defined as

$$Cost(Q_i, r_i) = \sum_{q \in Q_i} Cost(q, r_i) \tag{1}$$

The total processing time depends on the slowest replica:

$$Cost(Q, R) = max(Cost(Q_i, r_i)) \ , \ i \in [1, N] \tag{2}$$

The layout of the replica and the routing strategy considerably impact query performance. Therefore, finding an optimal layout of heterogeneous replicas and an appropriate routing strategy is essential. The MRCP is defined as follows:

**Definition 1.** *MRCP: Given a multi-dimensional dataset $P$ with $n$ dimensions, a query workload $Q$, and the replication factor $N$, we find a layout of heterogeneous replicas $R^*$ (the permutation of dimensions) and an adaptive routing strategy, such that the $Cost(Q, R)$ is minimized:*

$$R^* = arg \min_{R} \{Cost(Q, R)\} \tag{3}$$

**Hardness:** The MRCP is an NP-hard problem. Therefore, trying all possible permutations of the dimension columns is infeasible. For example, 3 replicas with 7 dimensions contain 128 billion permutations.

*Proof.* To analyze the hardness of MRCP, we first introduce the column ordering problem (COP) [2]: given a group of queries $Q$ and a set of columns in a column store, we find an optimal column ordering that has minimum query costs. This problem is NP-hard. Given a COP instance, we construct an MRCP instance with one replica. Each column in the column store is a dimension in our multi-dimensional data model. For each query in $Q$, we construct a query instance on this replica. Thus, finding an optimal layout such that the $Cost(Q, R)$ is minimized is equivalent to a COP instance. We reduce the COP to the MRCP with one replica in polynomial time. Therefore, the MRCP is NP-Hard.

# 4 Heterogeneous Replicas

In this section, we first model the query cost on multi-dimensional data. Then, we define the routing strategy and describe the construction of a near-optimal heterogeneous replicas layout.

## 4.1 Cost Model

First, we establish the query cost model on a replica with a specific layout. For a query, a range of data needs to be scanned, which is denoted as the **candidate set** *Row* and calculated as follows:

Given a layout of replica, $A = \{k_1, k_2, ..., k_n\}$, for each subquery $CNF$, suppose the $m$-th dimension in $A$ is the first dimension with a range predicate, i.e., (1) $\forall i \in [1, m-1]$, $\sigma(k_i)$ is an equality predicate; and (2) $\sigma(k_m)$ is a range predicate. Because predicates on $k_1$–$k_{m-1}$ are equality predicates, these predicates can form a prefix $(v_1, v_2, ..., v_{m-1})$. Rather than scanning all of the data items, the database can leverage the prefix to prune the data to be scanned. For predicate $\sigma(k_m) = \{p | p.k_m \in [s, e)\}$, we further derive a range of data items whose values in the dimension $m$ are between $s$ and $e$. Therefore, the database can quickly locate the two data items

$$p_s^A = (v_1, v_2, ...v_{m-1}, s_m, \_),\ p_e^A = (v_1, v_2, ...v_{m-1}, e_m, \_)$$

in which $p_s^A$ and $p_e^A$ are the first points whose values of the first $m$ dimensions are equal to the given values. It is difficult to use predicates on the latter dimensions $\{k_i | i > m\}$ to prune the scanned data. Therefore, the remaining values of the $n-m$ columns are omitted and denoted as "_". We call the two points **boundary points**. In the query process, all data items between $p_s^A$ and $p_e^A$ need to be scanned to derive the final result, which forms the candidate set.

We can measure the query cost by using the statistics of the dataset. Given a dataset $S$, $|S|$ is the number of data items in $S$. For each dimension $d_i$ in $P$, the **distribution function** of the value in $d_i$ is $F_{d_i}(x)$, and the **probability density function** is $f_{d_i}(x)$, where $f_{d_i}(x) = dF_{d_i}(x)/dx$. The major time cost of a subquery $CNF$ on replica $r$ depends on the candidate set $Row_r^{CNF}$ (i.e., the number of rows that need to searched), which can be estimated as follows:

$$Row_r^{CNF} = |S| \times \prod_{i=1}^{m-1} f_{k_i}(v_i) \times (F_{k_m}(e_m) - F_{k_m}(s_m)) \tag{4}$$

In this equation, we first estimate the number of data items by multiplying the probability density of each value in the equality predicate $f_{k_i}(v_i)$. Then, we multiply the proportion of data in dimension $k_m$: $F_{k_m}(e_m) - F_{k_m}(s_m)$. The total cost of a query instance $q$ on replica $r$ is

$$Row_r^q = \sum_{p=1}^{x} Row_r^{CNF_p} \tag{5}$$

The data that a query needs to scan may be smaller than what we estimate. It is possible to further reduce the number of data items in the estimate, by redefining $p_s^A$ as $(s_1, s_2, ...s_n)$, where $s_i$ is the value of the equality predicates or the start value of the range predicates on $k_i$. This complicates the estimate of the candidate set and is not especially beneficial. Therefore, we do not change our model further.

In the query process of a database, the system needs to locate the data, scan the candidate set on the disk, and apply predicates to the data. Finally, it transfers the result to the client through the network. We use function $t()$ to model the total cost of $q$ on replica $r$:

$$Cost(q, r) = t(Row_r^q) \tag{6}$$

The function $t()$ depends on the actual environment of the system, including the disk throughput, the size of each data item, and other configurations of the system, which should be modeled in a real system. Given specific data layouts of $N$ replicas and a query $q$, there are many routing strategies that can be applied. Our routing strategy is as follows. We route the query to the replica that has minimal cost. The minimal time cost of a query is defined as

$$Cost_{min}(q, R) = \{Cost(q, r) | \nexists r_j \in R, Cost(q, r_j) < Cost(q, r_i)\} \tag{7}$$

Although it seems that this might cause a load imbalance, our optimization goal helps to avoid this. Furthermore, we found that this strategy performed best with our proposed heterogeneous replicas construction algorithm.

### 4.2 Replica Construction

We here propose an algorithm, called Simulated annealing-based [2,9] Multi-dimensional data Replica Construction (SMRC), to find an approximation of the optimal heterogeneous replicas. Algorithm 1 provides details, in which a specific layout of all heterogeneous replicas $R$ corresponds to a **state**. The inputs include the query workload $Q$, an arbitrary state as the initial state $R_0$, and two parameters: the initial temperature $t_0$ and cooling_rate. In this algorithm, the temperature shrinks at a rate of $1 - cooling\_rate$. The SMRC returns an approximate optimal state $R$ such that the average query latency is minimized.

Lines 2–9 in Algorithm 1 are the main searching process in simulated annealing. A "good" state will always be accepted, whereas a "bad" state will be accepted probabilistically to avoid the local optimum. The new state generation function $NewState(R)$ is implemented as follows. We randomly choose two columns $k_i$ and $k_j$, and apply one of the following three operations:

- $swap(k_i, k_j)$: swap the two columns $k_i$, $k_j$.
- $inverse(k_i, k_j)$: invert the columns between $k_i$ and $k_j$.
- $insert(k_i, k_j)$: insert $k_i$ into the position of another column $k_j$.

**Algorithm 1.** Simulated annealing-based Multi-dimensional data Replica Construction (SMRC)

**Input:**
$Q$: Query workload
$R_0 = \{r_1, r_2, ...r_N\}$:Initial structure of replicas
**Output:**
$R$: Optimized heterogeneous replicas
1: $t := t_0, R := R_0, C := Cost(Q, R_0)$
2: **for** $k = 1$ *to* $k_{max}$ **do**
3: $t := t \times (1 - cooling\_rate)$
4: $R' := NewState(R)$
5: $C' := Cost(Q, R')$
6: **if** $C' < C || e^{\frac{C-C'}{t}} > random(0, 1)$ **then**
7: $R := R'$, $C := C'$
8: **end if**
9: **end for**
10: **return** $R$

Although inverse and insert could be replaced with multiple swap operations, we retain them for fast traversal. When $NewState(R)$ is called, it performs one of the above operations on a random selected replica $r_j$ in $R$. We illustrate the state generation process using three operations above in Fig. 2. The initial state is $R_0$. We then use $swap(k_3, k_2)$ on $r_2$ to generate a new state $R_1$. Suppose that all new states are better in this example. Then, the current state is $R_1$, and we use $Inverse(k_1, k_4)$ on $r_1$ to generate $R_2$. Finally, we use $Insert(k_2, k_1)$ on $r_3$ to generate state $R_3$.

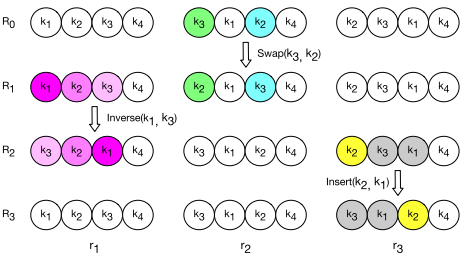



**Fig. 2.** Example of state generation using three operations

## 5 Experiments

### 5.1 Implementation

We implemented our method on Apache Cassandra, which is widely used at Hulu, GitHub, eBay, and over 1500 companies for mission-critical data.

In Cassandra, each tuple contains a partition key and some clustering keys. The partition key is used to partition data across the nodes. Clustering keys are the dimensions of data. Cassandra serializes data items according to the order of the clustering column values. By controlling the order of clustering columns, we can control the data layout on the disk. We did not change the partition key, and we optimized the data layout in each partition. For a table with $n$ replicas, we generated $n$ replica layouts by SMRC and constructed the heterogeneous replicas in Cassandra. All writes and queries were routed by a middle layer that deployed a request router.

### 5.2 Experimental Methodology

**Hardware:** The experiments were carried out on a Cassandra (version 3.11.4) cluster with 5 nodes. Each node had 2 Intel Xeon E5-2697 CPUs with 36 cores in total, 8 GB memory, and a 7200-rpm HDD. The operating system was 64-bit Ubuntu Server 16.04.4 LTS with Linux kernel version 4.15.0-36.

**Comparison:** We compared our method to two state-of-the-art methods:

- NoSE [11]: In this strategy, all replicas had the same layout, which can be seen as the best layout for uniform replicas.
- Divergent Design (DIVG) [4]: In this method, the **balance factor** $m$ (each query cost was evenly shared by $m$ replicas) ranged from 1 to $N-1$, where $N$ is the replication factor. When $m = N$, DIVG was equal to NoSE because each query was routed to $N$ replicas. Thus, we omitted this case. When designing each replica layout, the DBAdv() was our method under one replica.
- Simulated Annealing based Multi-dimensional data Replica Construction (SMRC): With SMRC, $k_{max}$ was set to 1000, and *cooling_rate* was set to 0.1 to ensure that the temperature cooled down to nearly zero for a near-optimal result. 100 budgets were used in the histogram for each dimension.

**Dataset**: There were two datasets:

- TPC-H: We used the "lineitem" table in TPC-H. The "lineitem" table has 17 columns, in which 7 columns are the dimensions (*l_quantity*, *l_partkey*, *l_orderkey*, *l_linenumber*, *l_extendedprice*, *l_suppkey*, *l_discount*). We used a scale factor of 20–100, resulting in different data sizes, ranging from 120 million to 600 million data items in the table.
- Synthetic dataset: To simulate big data, we generated a synthetic dataset that contained 1000 dimensions. The value scope of each dimension was 1–100. The data was uniformly distributed in the space. The total number of data items was 1 billion.

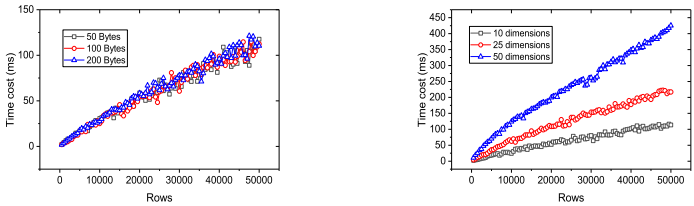
**Workload:** Two workloads were generated for each dataset. Each workload contained 1000 query instances. Supposing that the dataset has $n$ dimensions, each query instance contains one CNF with $n$ predicates. The predicate is either a full

range predicate or an equality predicate. The selectivity of a range predicate is 10% on its dimension. We generated a skewed workload $Q_s$ where some dimensions had more range predicates than others. We also generated a uniform query workload, called $Q_u$, where the range predicates were uniformly distributed on each dimension.

**Metrics:** We used $Cost(Q, R)$ to measure the performance of the different methods. The lower $Cost(Q, R)$ is, the higher the query throughput is. To guarantee accuracy, we ran each algorithm three times and used the average values.

In the following experiments, we answer six questions:

1) How can we get the cost model in a real cluster? Sect. 5.3
2) How does our method perform under different circumstances? (Sect. 5.4).
3) What additional cost will heterogeneous replicas bring? (Sect. 5.5).
4) Can simulated annealing be replaced by other heuristic algorithms? (Sect. 5.6).
5) What is the SMRC's performance when facing node failure? (Sect. 5.7).




(a) with a different size of value column (b) with a different number of clustering columns

**Fig. 3.** Modeling $t()$ on Cassandra

### 5.3 Model Robustness

Deriving an exact cost function is challenging, because the cost function not only depends on the configurations of the system but also on hardware performance. Therefore, we treated it as a black box and evaluated the function $t()$ statistically. We found that $t()$ is a linear function under different workloads.

We generated a number of queries where the size of the candidate set $Row_r^q$ differed on a simulation dataset. We enabled **tracing** to profile the time cost in each query using the *"tracing on"* command[1] and recorded the query latency.

We evaluated the time cost function $t()$ with different sizes of data items on a synthetic dataset with ten dimensions. First, we changed the size of the metrics columns to control the size of each data item from 50 bytes to 200 bytes. Figure 3(a) shows the result. The points show the roughly linear relationship

[1] https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlshTracing.html.

between candidate set $Row_r^q$ and the time cost. The cost did not change significantly when the size of the data items increased.

Second, we studied the impact of the number of dimensions on the cost function. We changed the number of dimensions from 10 to 50. As can be seen in Fig. 3(b), the cost function remained linear with different numbers of clustering columns. Unlike the increasing size of the value column, the slope of the cost function increased considerably when the number of dimensions increased. Thus, the cost function $C(q, r)$ can be replaced by $Row_r^q$ without impacting the result of the SMRC algorithm.

### 5.4 Performance of SMRC

The query time cost of the different methods on the TPC-H dataset is shown in Fig. 4. The scale factor was 20, the replication factor was 3, and the workload was $Q_s$. With DIVG, we used a balance factor $m$ of 1, because we found that a higher $m$ value degraded the query throughput of DIVG. For example, when $m$ was 1, the Cost(Q, R) was 26,800 s, whereas it was 54,800 s when $m$ was 2. The reason for this is that when the parameter $m$ is set higher, more query types need to be served by a replica, resulting in less diversity in each replica. We set balance factor to 1 for DIVG in the following experiments.

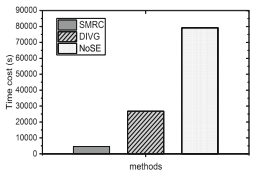



**Fig. 4.** Time cost with $Q_s$ on the TPC-H dataset

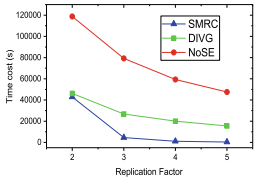



**Fig. 5.** Replication factor with $Q_s$ on the TPC-H dataset

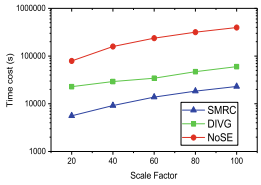



**Fig. 6.** Data size with $Q_s$ on the TPC-H dataset

To evaluate the performance and scalability of the different methods further, we performed the following experiments under different replication factors, different dataset scales, and different workloads.

**Replication Factor:** The benefits of heterogeneous replicas depend on the replication factor. For most applications, the replication factor is between two and five to support high availability when facing node failure. We evaluated the impact of the replication factor on query time cost with the TPC-H dataset and workload $Q_s$. The scale factor was 20, and the results are shown in Fig. 5. When the replication factor was two, SMRC was similar to DIVG, because the alternative replica layouts were limited and both methods could find a good state. As the replication factor increased, however, the number of potential layouts grew

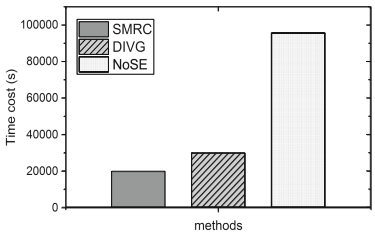



**Fig. 7.** Workload $Q_u$ on the TPC-H dataset

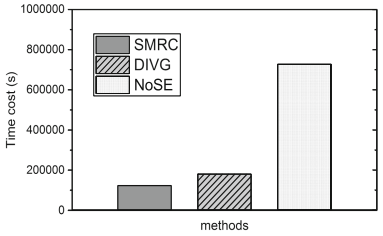



**Fig. 8.** Large-scale problem with $Q_s$ on the synthetic dataset

exponentially. The advantage of SMRC is more obvious. Furthermore, even two heterogeneous replicas greatly improved query throughput compared to NoSE.

**Data Size:** We evaluated query time cost on the TPC-H dataset with a scale factor from 20 to 100, and a replication factor of 3. Figure 6 shows the time cost of the different strategies. As the size of the dataset increased, the size of the result set grew proportionally. Under different scale factors, SMRC always required the least time among the methods, demonstrating that SMRC has good scalability.

**Uniform Workload:** The performance of workload $Q_u$ on the TPC-H dataset is shown in Fig. 7. The scale factor was 20 and the replication factor was 3. Compared to the skewed workload, the gap between SMRC and DIVG narrowed. For a uniform workload, DIVG was more likely to find a near-optimal solution, which means that query clustering was more suitable for that workload. However, SMRC still performed better than DIVG.

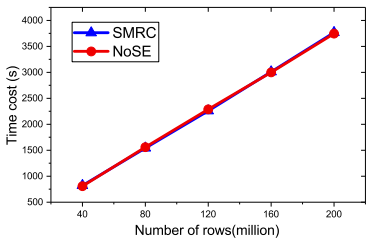



**Fig. 9.** Insertion time of the TPC-H dataset

**Problem Scale:** The scale of the problem depends on the searching space, except for the TPC-H dataset with seven dimensions. We also used the synthetic dataset with 1000 dimensions. The replication factor was 3 and the query cost of the different methods is in Fig. 8. Even with hundreds or thousands of dimensions, SMRC performed the best. Thus, SMRC can solve large-scale problems.

### 5.5 Overhead of Heterogeneous Replicas

**Write Speed:** We evaluated the writing speed of heterogeneous replicas by loading the TPC-H data into different layouts generated by SMRC and NoSE. In real applications, the data insertion order is difficult to guarantee. We loaded the data in order of origin. The result is shown in Fig. 9. Our evaluation shows that heterogeneous replicas bring no additional cost to the insertion speed, because we do not maintain any extra structure or dataset.

**Memory Consumption and Routing Overhead:** The memory consumption of our method is low. Unlike [18], we do not store the training workload. Only the statistics of the data are kept in memory. For a dataset with 7 dimensions, if 100 budgets (each budget stores the value count) are used in each histogram, the total memory usage is 5 KB ($7 \times 100 \times 8B$). When routing a query, we only need to execute $N$ calculations to find the most efficient replica, where $N$ is the replication factor. In our experiments, the routing time occupied less than 0.5% of the query processing, because they are in-memory calculations.

### 5.6 Other Heuristic Algorithms

In addition to simulated annealing, we also attempted to use the genetic algorithm [20], insofar as it is also suitable for multi-dimensional data replica construction. However, the genetic algorithm required more time to converge—approximately five times more than SMRC. Although other heuristic algorithms may work, we focused on SMRC in this paper for efficiency.

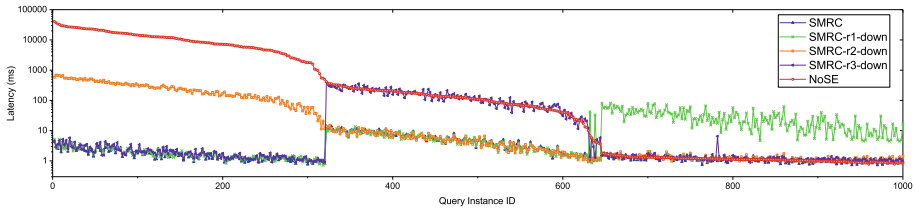



**Fig. 10.** Latency of each query instance when facing node failure (Color figure online)

### 5.7 Failure and Recovery

To explore the behavior of our method when facing failure, we conducted the following experiments. The dataset we used was TPC-H, the workload was $Q_s$, and the replication factor was set to three.

Replication is used to guarantee system availability. Given three replicas, r1, r2, and r3, we can choose only two replicas to route queries when a node is down. Intuitively, many queries may be worse than uniform replicas in this

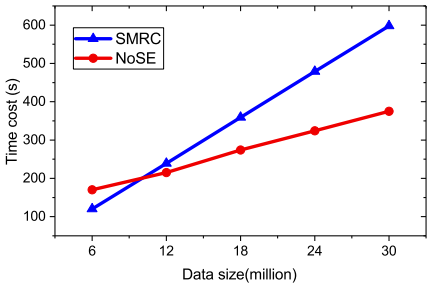



**Fig. 11.** Data recovery speed

situation. We traced the latency of each query in this scenario, and the results are shown in Fig. 10. The query instances are sorted according to descending order of latency on NoSE (red line). The blue line represents all replicas that were alive in SMRC. In this case, each replica served approximately a third of the query instances. Query instances 0–320 were routed to r2, 321–632 to r3, and 633–1000 to r1. The other three lines represent each replica that is down in SMRC. Obviously, when a replica is down, approximately a third of the queries will be affected. However, the query latency was still acceptable and the query time cost was greater than NoSE. Improvements to query time cost in the case of node failure were discussed in [18], and the solutions offered there can be applied to our method.

We recover a replica by rewriting the data to another replica. We measured the recovery speed of NoSE with the repair command in Cassandra, and we measured the recovery speed of SMRC with rewritten data. For NoSE, we first stopped one node in a Cassandra cluster and removed that node's data folder. Then, we restarted the Cassandra process and called the *nodetool repair -full* to launch the data recovery process in Cassandra. The result is shown in Fig. 11. The recovery speed of NoSE was 60% faster than that of the proposed SMRC. However, disk failures occur infrequently, and recoverability is more important than recovery speed. Thus, considering the tremendous improvement in query performance, the speed of recovery with SMRC is acceptable.

When the query workload changes dramatically, query throughput can decrease with current replica layouts. When the query performance decreases to a certain threshold, the algorithm can generate a new replica layout for restructuring data. There are two ways to restructure the data: (1) restructuring all historical data [18], or (2) restructuring recent data [2]. These approaches are both applicable to our methods.

## 6 Conclusions

In this paper, we studied and defined the problem of how to leverage heterogeneous replicas to improve query throughput on multi-dimensional data, which is

widely used in applications. Existing approaches to accelerate queries either optimize limited kinds of queries by adjusting the layout of the data, or they bring additional overhead by introducing auxiliary structures. The proposed method, however, does not introduce any additional disk cost when optimizing the existing replica layout on the disk.

We modeled the query cost with multi-dimensional data, and introduced a routing strategy and a replica construction algorithm. The proposed method outperformed state-of-the-art solutions. Furthermore, our solutions did not incur additional overhead, such as extra disk occupation or slowed insertion speeds. When node failures occurred, our heterogeneous replicas worked well and could recover in a reasonable time. We believe that our solutions can be easily applied to other multi-dimensional data management systems apart from Cassandra. The future work is finding some rules to accelerate the searching of SMRC algorithm.

**Acknowledgments.** The work was supported by the Nature Science Foundation of China (No. 61802224, 71690231), and Beijing Key Laboratory of Industrial Bigdata System and Application. We also thank anonymous reviewers for their valuable comments.

## References

1. Ailamaki, A., DeWitt, D.J., Hill, M.D., Skounakis, M.: Weaving relations for cache performance. VLDB **1**, 169–180 (2001)
2. Bian, H., Yan: Wide table layout optimization based on column ordering and duplication. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 299–314. ACM (2017)
3. Borthakur, D., et al.: HDFS architecture guide. Hadoop Apache Project 53 (2008)
4. Consens, M.P., Ioannidou, K., LeFevre, J., Polyzotis, N.: Divergent physical design tuning for replicated databases. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 49–60. ACM (2012)
5. Copeland, G.P., Khoshafian, S.: A decomposition storage model. In: SIGMOD Conference (1985)
6. Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudré-Mauroux, P., Madden, S.: Hyrise - a main memory hybrid storage engine. PVLDB **4**, 105–116 (2010)
7. Jindal, A., Quiané-Ruiz, J.A., Dittrich, J.: Trojan data layouts: right shoes for a running elephant. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, p. 21. ACM (2011)
8. Jouini, K.: Distorted replicas: intelligent replication schemes to boost I/O throughput in document-stores. In: 2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA), pp. 25–32 (2017)
9. Kirkpatrick, S., Gelatt, D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**, 671–680 (1983)
10. Lamb, A., et al.: The vertica analytic database: C-store 7 years later. PVLDB **5**, 1790–1801 (2012)
11. Mior, M.J., Salem, K., Aboulnaga, A., Liu, R.: NoSE: schema design for NoSQL applications. IEEE Trans. Knowl. Data Eng. **29**(10), 2275–2289 (2017)
12. Home page P (2018). http://parquet.apache.org/documentation/latest/

13. Rabl, T., Jacobsen, H.A.: Query centric partitioning and allocation for partially replicated database systems. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 315–330. ACM (2017)
14. Ramamurthy, R., DeWitt, D.J., Su, Q.: A case for fractured mirrors. VLDB J. **12**, 89–101 (2002)
15. Saccà, D., Wiederhold, G.: Database partitioning in a cluster of processors. ACM Trans. Database Syst. **10**, 29–56 (1983)
16. Staudt, M., Jarke, M.: Incremental maintenance of externally materialized views. In: VLDB (1996)
17. Stonebraker, M., et al.: C-store: a column-oriented DBMs. In: Proceedings of the 31st International Conference on Very Large Data Bases, pp. 553–564. VLDB Endowment (2005)
18. Tran, Q.T., Jimenez, I., Wang, R., Polyzotis, N., Ailamaki, A.: RITA: an index-tuning advisor for replicated databases. In: Proceedings of the 27th International Conference on Scientific and Statistical Database Management, p. 22. ACM (2015)
19. Valentin, G., Zuliani, M., Zilio, D.C., Lohman, G., Skelley, A.: DB2 advisor: an optimizer smart enough to recommend its own indexes. In: Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073), pp. 101–110. IEEE (2000)
20. Whitley, D.: A genetic algorithm tutorial (1994)
21. Xiang-dong, H., Jian-min, W., Si-han, G., et al.: A storage model for large scale multi-dimension data files. Proc NDBC **1**, 48–56 (2014)
22. Xu, C., Tang, B., Yiu, M.L.: Diversified caching for replicated web search engines. 2015 IEEE 31st International Conference on Data Engineering, pp. 207–218 (2015)