Apara: Workload-Aware Data Partition and Replication for Parallel Databases

Xiaolei Zhang[2], Chunxi Zhang[2], Yuming Li[2], Rong Zhang[1,2(✉)], and Aoying Zhou[2]

[1] International Research Center of Trustworthy Software, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
[2] School of Data Science and Engineering, East China Normal University, Shanghai, China
xiaoleizhang.ecnu@gmail.com, {cxzhang,liyuming}@stu.ecnu.edu.cn, {rzhang,ayzhou}@dase.ecnu.edu.cn

Abstract. Data partition and replication mechanisms directly determine query execution patterns in parallel database systems, which have a great impact on system performance. Recently, there have been some workload-aware data storage techniques, but they suffer from problems of narrow support to complex workloads or large requirements for storage. In order to enable the support for complex analytical workloads over massive distributed database systems, we design and implement a workload-aware data partition and replication tool, called *Apara*. We design two heuristic algorithms and define two cost models for effective data partition calculation and efficient replication usages. We run a set of experiments to compare and demonstrate the performance between *Apara* and the other representative work. The results show that *Apara* consistently outperforms the primary solutions on TPC-H workloads.

Keywords: Distributed database · Workload-aware storage · Partition · Replication

1 Introduction

As the explosion of data and severe requirement of massive query processing ability, parallel database systems and parallel data processing platforms are developed. Generally, they horizontally partition large amounts of data to distributed nodes in order to provide parallel data processing capabilities for analytical queries. One of the major challenges when horizontally partition data is to achieve low data transferring for executing analytical queries [4,9,14,20]. Data partition and replication are the main technology to reduce the processing cost for those analytical workloads, which shall guarantee process parallelization and data locality. The traditional approach splits each table on some key, using hash or range partition. Hash partition is good for the point query, and range partition

J. Shao et al. (Eds.): APWeb-WAIM 2019, LNCS 11642, pp. 191–206, 2019.
https://doi.org/10.1007/978-3-030-26075-0_15

makes data within a given range of the partition key in the same partition. This helps queries that have selection predicates involving the key go faster, but does not affect the performance of queries without the split key attribute. we demonstrate the critical impact of remote data transferring on query performance in Fig. 1. The test workload is Q_3 in benchmark TPC-H [17], shown in Fig. 2. This experiment is conducted on Greenplum which is deployed on 9 nodes equipped with the Gigabit network (more details presented in Sect. 5). Traditional hash partition based on primary keys or randomly selected attributes have much worse performance compared to our Apara. *Apara* can significantly improve the query performance by 82.2% compared to key-based partition scheme.

Traditionally, hash or range partitioning to data can improve the performance of queries involving the key in selection predicates. For join operators, queries will benefit by co-partition technology on the join attributes. However this method may not be suitable for complex scheme, which can only be used to a subsets of tables sharing the join keys. Oracle [4] proposes a reference partition method REF, and it co-partitions a table by another table that is referenced by an outgoing foreign key. It can avoid duplicating the partition key columns and improve the data integrity. But it cannot be used to optimize the network transmission in distributed environment. Predicate-based reference partition method PREF [20] is a partition scheme that allows to co-partition tables based on given join predicates. However, if there is a deep cascading reference relationship in the schema, substantial data redundancy will be stored in child tables. AdaptDB [9] is a work of adaptive data partition. But it has a strong assumption that reading a remote disk is similar as a local disk and it is unrealistic currently.

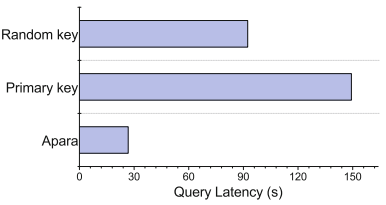


Fig. 1. Query latencies with different partition schemes

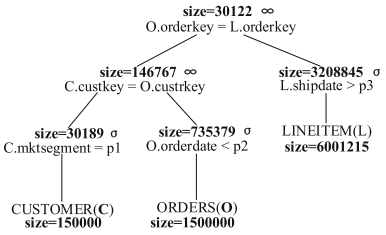


Fig. 2. Q3 in TPC-H

In this page, we propose *Apara*, a workload-aware distributed partition and replication tool enabling data distribution effectively and efficiently for parallel database systems. With the input of target application workloads and corresponding database schema, *Apara* can find an appropriate storage mode tailored for the application by near-optimal algorithms. In summary, we make the following major technical contributions:

– We propose a workload-aware data partition tool *Apara* with data partition and replication mechanisms for complex analytical workloads. It is the first

work to optimize the data transfer cost for production environment, which can support the complicated workload with the multiple TPC-H style queries.
– We design two near-optimal algorithms which are greedy algorithm and genetic algorithm for partition configuration generation with detailed analysis for the efficiency of our algorithms. We define two cost models, which consider data transferring cost and data replication cost, respectively. It is the first work to limit data redundancy in considering optimizing query processing performance.
– We present extensive experimental results to show that *Apara* has excellent performance on TPC-H workloads and outperforms the other methods.

The rest of this paper is organized as follows. Section 2 describes the overview of Apara. Section 3 presents the methods designed for data partition. Section 4 shows the details of the cost models for both network and replication. Section 5 gives an experimental study. Section 6 describes some related work and Sect. 7 concludes the paper.

2 Overview of Apara

In this paper we provide an workload-Aware data PArtition and ReplicAgtion tool (Apara for short) for complex join processing with the purpose of finding the optimal partition strategy for each table. Apara can be used as a peripheral tool or embedded in the storage layer as a part of the physical design of the database.

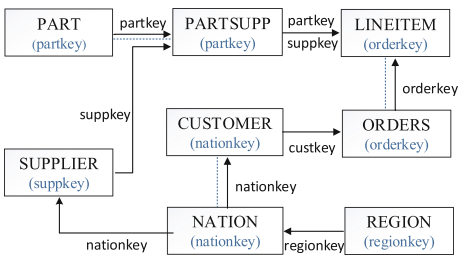


Fig. 3. Schema reference in TPC-H

2.1 Preliminary and Definition

Through the paper, we take the analytical queries in TPC-H as example. The reference relationship and a simplified partition schema among tables are drawn in Fig. 3. The solid arrow line stands for the reference relationship among tables and dotted line is an example of partition configuration. We can see that if table **ORDERS** is hash partitioned on its primary key *orderkey*, then table **LINEITEM** can be co-partitioned using the outgoing foreign key *fk* to

ORDERS. Apara uses hash partition method by default for tables to ensure that the data distributed to each node is roughly balanced. In order to make the description easy, we use capital letter S, W and T to represent schema, workload set and tables. We use A and a to represent attribute set and any single attribute respectively.

Definition 1. *Partition Attribute Set A^T: For table T, there are m join attributes $\{a_k\}$ $(0 \leq k \leq m-1)$. We call these m attributes partition attribute set for T, represented as A^T.*

One table may join with several other tables having m join attributes, but there is only one attribute a, $a \in A^T$ for partition. For example, in Fig. 3, **NATION** joins with **SUPPLIER**, **CUSTOMER** and **REGION**. Then $A^{NATION} = \{nationkey, regionkey\}$.

Definition 2. *Partition Configuration P : P is a collection of pairs like $<T, a>$, where for table T, it selects attribute $a, a \in A^T$ as its partition attribute. If we have n tables, then $|P| = n$.*

Definition 3. *Problem Definition: Given a schema S involved in workload W, find a good partition P for W such that network transferring $C(P)$ is minimized,i.e.,*

$$\begin{aligned} & arg_p min \quad C(P) \\ & subject\ to \quad select\ a\ for\ T, \\ & \qquad\qquad \forall\ a \in A^T, \forall\ T \in S. \end{aligned}$$

Apara is designed to find a partition configuration P defined in Definition 2, which can help to reduce data transferring cost in distributed environment defined in Definition 3.

2.2 Cost Model

Data transferring among distributed nodes costs the most for distributed query processing. Data partition quality in distributed environment determines the amount of data transferred remotely. The amount of data transferred between nodes is then used to evaluate the data locality.

$$cost_D^P(W) = \sum_{q_i \in W} Cost_D^P(q_i) \tag{1}$$

Where $cost_D^P(W)$ is the total cost of data transferring for workload W under partition configuration P and $Cost_D^P(q_i)$ is the cost for query q_i decided by its involved join operations.

For a cluster with M nodes, the data transferring cost for $T_A \bowtie T_B$ is calculated as follows:

$$Cost_D^{T_A \bowtie T_B} = \begin{cases} 0, & \text{if } co-partition \\ min(S_A, S_B) \times \frac{M-1}{M}, & \text{if } shuffle\ join \\ min(S_A, S_B) \times (M-1), & \text{if } copy-based \end{cases} \tag{2}$$

Where S_A and S_B are the data of T_A and T_B that takes part in the join operation. For co-partition, there is no additional data transferring. If we take shuffle join, we should shuffle the data of small table in one node to all other $(M-1)$ nodes. Supposing S_A is the smaller one and each node stores $\frac{1}{M}$ parts of S_A, when shuffle starts, the shuffling data size of each node is $\overline{D}_{shuffle} = \frac{M-1}{M} \times \frac{|S_A|}{M}$ and the total shuffling size is then $D_{shuffle} = \overline{D}_{shuffle} \times M = \frac{M-1}{M} \times |S_A|$. If table is small enough, we can just copy it to the other $M-1$ nodes.

2.3 Apara Architecture

Apara is designed sensitive to the changes in underlying workloads by enabling the distributed database system to partition and replicate data for improving the join performance. Figure 4 shows the main components of Apara. Inputs to Apara are the database schema and historical query workloads expressed as the query trees. We provide three different tunning algorithms for data distribution, which are optimal partition, greedy partition and genetic partition algorithms. The partition strategy is evaluated by our cost models, which consider data transferring without replication cost (Network-based Cost) and data transferring with replication cost (Network and Replication Cost) respectively. Finally the partition and replication configuration are generated.

From the input database schema, we can get all the table information, e.g. table name and table size. The historical executed query workloads are abstracted as query trees. Each query tree generally involves multiple join operations. Each join operation in the tree has the information about the two join tables, the join attributes, the filter conditions, filter ratio of each table, and the size of join intermediate result set. An example of query tree is shown in Fig. 2, where the number is the data size.

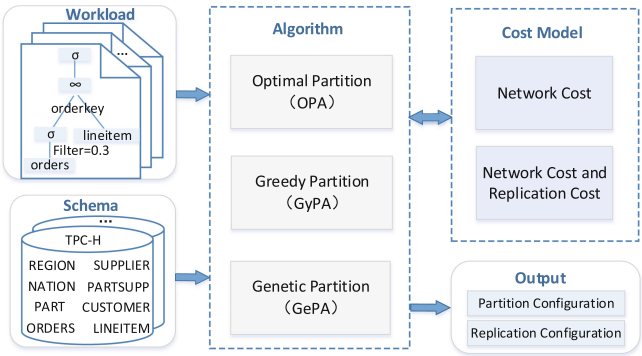


Fig. 4. Overview of Apara architecture

The Optimal Partition module **OPA** tries a traversal search method to find the optimal data partition strategy. But when the number of tables involved with

join operations in W is large, it is impossible to traverse all possible candidate partitions because this is a NP-hard problem. So we design new heuristic search strategies to solve this problem. Greedy Partition algorithm **GyPA** is designed by the guide of the data transferring cost among all joins in W, and the Genetic Partition algorithm **GePA** is designed to seek the potential optimal data partition configuration by mapping data partition problem to genetic evolution of nature species. Besides data partition, replication is another way for enhancing data locality. However, copying all the data to the nodes involving with them is obviously unreasonable, because it will generate a lot of redundancy. Then the number of copies and redundant usage of storage resources should be controllable.

3 Workload-Aware Partitioning Algorithm

3.1 Optimal Partition Algorithm *OPA*

The simplest search algorithm is to traverse all partition candidates and then select the partition configuration P with the least transfer cost as shown in Algorithm 1, where we have n tables for partition and each table has its own partition attributes in *PAS*. Line 4–11 is the recursive code for finding all the partition candidates in *allPartitionConfig*. After having all the partition candidates, Line 12 computes the data transferring cost $Cost_D^P$ of every partition configuration P and Line 13 selects the partition with the least cost as the optimal partition configuration. There are N tables involved in join operations with $|P| = N$. The number of average potential partition key of each table is M. Then the time complexity of the traversal search is $O(M^N)$. Clearly when there are many tables in the workload, the search space will be too large to traversing all the partition candidates in limited time. Therefore, we design following heuristic algorithms to solve this problem.

Algorithm 1. Optimal Partition Algorithm: *OPA*

Input: historical workload W, partitionAttributeSets *PAS*
Output: partition configuration P

1 *allPartitionConfig* $\leftarrow \emptyset$;
2 *partitionConfig* $\leftarrow \emptyset$;
3 $n = |PAS|$, i = 0;
4 getAllParCandidate(*allPartitionConfig*, i, *partitionConfig*){
5 **if** $i == n$ **then**
6 add(*partitionConfig*, *allPartitionConfig*); % find one candidate
7 return ;
8 **for** *(j = 0; j* $< |A^{T_i}|, A^{T_i} \in PAS$*; j++)* **do**
9 select one partition attribute $a_j^{T_i} \in A^{T_i}$ into *partitionConfig*;
10 getAllParCandidate(*allPartitionConfig*, i++, *partitionConfig*);
11 }
12 computer $Cost_D^P, P \in$ *allPartitionConfig* for each partition candidate;
13 return *partitionConfig* with the least cost;

3.2 Greedy Algorithm for Partitioning *GyPA*

For join between two tables in distributed environment, the network transferring cost will be different according to the selected partition attributes and the join attributes. Generally there are five kinds of join options summarized in Table 1, which are left table shuffle(**LS**), right table shuffle(**RS**), co-partition(**CP**), left table shuffle plus right table shuffle(**LSRS**), table copy(**TC**). Let's take a look at the join segment "*select * from NATION,REGION where NATION.regionkey = REGION.regionkey*" for example, involving table NATION (size = 15) and REGION (size = 5) with join attribute *regionkey*. Selecting different partition attributes, the join costs are different as calculated in Eq. 2, with $M = 5$ as the number of involved distributed nodes:

1. The partition attribute is *regionkey* for both NATION and REGION. Two tables can be co-partitioned with minimal cost *0*;
2. The partition attributes for NATION and REGION are *nationkey* and *regionkey*, respectively. If we perform a shuffle join in NATION according to *regionkey* attribute, the cost of shuffle join **RS** is $\frac{M-1}{M} * |Nation| = \frac{4}{5} \times 15 = 12$. However, if we broadcast REGION to all other nodes, the **TC** broadcast cost is $(M-1) \times |Region| = 4 \times 5 = 20$. Shuffle join is preferred in this case.
3. The partition attributes of NATION and REGION are *regionkey* and any non-key attribute, e.g. *name*, respectively. If we perform **LS** shuffle join in REGION according to *regionkey*, the cost is $\frac{M-1}{M} \times |Region| = \frac{4}{5} \times 5 = 4$. However, when NATION is broadcast to all other nodes, the **TC** broadcast cost is $(M-1) \times |Nation| = 4 \times 15 = 60$. Obviously, we should shuffle REGION table instead of copying the NATION table.
4. Neither of the partition attribute of NATION or REGION is the join attribute, e.g. *nationkey* for NATION and *name* for REGION. If we perform **LSRS** shuffle join both in REGION and NATION according to *regionkey* attribute, the cost is $\frac{M-1}{M} \times |Nation| + \frac{M-1}{M} \times |Region| = \frac{4}{5} \times 15 + \frac{4}{5} \times 5 = 16$. If one of the two tables is broadcast to the other nodes, the **TC** cost is $min((M-1) * |Nation|(M-1) * |Region|) = min(60, 20) = 20$. Obviously, we should copy REGION table instead of shuffling the two tables.

Table 1. Example Cost Table for R, S and T (where '−1' is the unavailable partition)

(a) Initial Table

cost / join	LS	**RS**	CP	LSRS	TC
R.c2, S.c2	4	12	0	-1	5
S.c2, T.c2	12	**32**	**0**	-1	15

(b) Updated Table

cost / join	LS	RS	CP	LSRS	TC
R.c2, S.c2	4	-1	0	-1	5
S.c2, T.c2	-1	-1	-1	-1	-1

Join cost can be calculated in advance using query trees and database schema. In order to reduce the cost during SQL execution as much as possible, we should

try to filter the expensive join ways as early as possible. So the greedy strategy **GyPA** is to fill the join cost table shown in Table 1, avoid the costly join strategy and select the least cost join method as far as possible.

Algorithm 2 gives the pseudo-code for our greedy algorithm. In Table 1, we have three tables **R**, **S**, and **T**; we have two joins $j_1 = R \underset{R.c_2=S.c_2}{\bowtie} S$ and $j_2 = S \underset{T.c_2=S.c_2}{\bowtie} T$. We first initialize the join cost table *JoinTable* in Line 3, shown in Table 1(a). For the join tables, we iterate (Line 4–8) to find the partition attributes for them. First, we decide the table for checking by finding the largest join cost in the *JoinTable*, which is the worst partition on key $c2$. In Table 1(a), the max cost is 32 by **RS** for j_2. For j_2, partition method **CP** can have the least cost 0. We can then update the *partitionConfig* by adding pairs $<S, c2>$ and $<T, c2>$ as in Line 7. For the selected partition attributes, we recalculate and update *JoinTable* and get a new one shown in Table 1(b) as in Line 8. In distributed environment, we may need to partition the other tables, which are not involved in any joins and are then tackled by hashing acquiescently on their primary key in Line 9. Notice that we select the attribute in partition table instead of the shuffle table to update *partitionConfig*. For example, if LS generates the least cost, the second table and its partition attribute pair is selected to join *partitionConfig*. If CP generates the least cost, both table and partition attribute pairs should be inserted to *partitionConfig*.

Algorithm 2. Greedy Algorithm for Partition: *GyPA*

Input: historical workload W, schema S
Output: partition configuration P

```
1  partitionConfig ← ∅;
2  JoinTable ← ∅;
3  fillJoinTable(JoinTable, W, S);
4  while !isAllUpdate() do
5      Key = getMaxCost(JoinTable);
6      Key = getMinCostByKey(JoinTable, Key);
7      add Key to partitionConfig;
8      update(JoinTable);
9  process the other tables;
10 return partitionConfig;
```

Supposing the size of *JoinTable* is n. In cost computing phase, we need traverse all the lines of *JoinTable*, so the time complexity is $O(n)$. When we locate the maximum cost and do updation to *JoinTable*, we may update all the other lines of the table, and the time complexity is $O(n^2)$. So the time complexity of *GyPA* is $O(k * n^2)$, where k is the number of tables involved with joins. But in fact, the method always makes the best choice in current view which is a local optimal solution and does not consider global optimization.

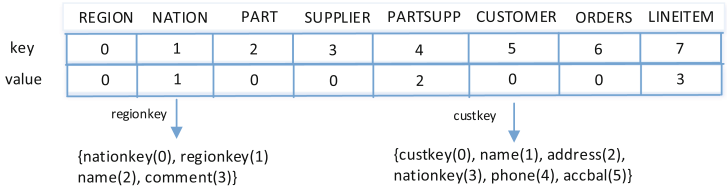


Fig. 5. Example individual for TPC-H

3.3 Genetic Algorithm for Partitioning *GePA*

We design a new heuristic algorithm based on the traditional genetic algorithm [19] shown in Algorithm 3 to find a global optimal partition configuration. First, we do individual encoding and population initialization in Line 2, where each individual corresponds to a partition configuration and the genotype of individual is the partition attribute selected for each table. Individual is represented as an array. We represent the individual for TPC-H as an eight-size array, indexed from 0 to 7 for the eight tables (key part). The value for each item is the index of the selected partition attribute, shown in Fig. 5. For example, NATION has four join key candidates, which are *nationkey, regionkey, name, comment*. The value for NATION is *1*, representing *nationkey* selected as its partition key. So Fig. 5 gives an example individual value as {01002003}. The number of individuals defined in the initial population is left as an adjustable parameter according to the scale of the problem. The given initial population size in this paper is 10.

Definition 4. *Selection Probability: Supposing the population size is N and one optional partition configuration is P_i, the Selection Probability of $P_i = \frac{Cost_D^i}{\sum_{i \in N} Cost_D^i}$.*

In the context of our work, the size of data transferring under each partition configuration is the indicator to measure the quality of individuals. We take the size of data transferring as the fitness function for gene evolution, which is also known as the evaluation function and used to select the superior and eliminate the inferior ones in population. The larger the data transferring $Cost_D$, the worse the individual fitness and the more likely to be eliminated. Choosing good partition configurations from the population and eliminating bad ones is executed in each round of iteration shown in Line 4–8 in Algorithm 3. We implement Roulette Selection for gene evolution. Roulette Selection does gene evolution based on Selection Probability defined in Definition 4, which reflects the ratio of the fitness of P_i to the total individual fitness of the entire population. The higher the P_i, the higher the probability of being selected as a bad individual. Line 6 is Crossover Operation, which can generate new partition configurations by randomly exchanging some genes of two individual in the population according to the crossover mode. CrossOne, also known as simple crossover, refers to the random setting of an intersection point in the individual gene string, then

randomly selecting two individuals as the parent individuals, exchanging the part of gene block behind their intersection point, and then generating two new child individuals. Line 7 is Variation Operation, which generates a new partition configuration with the possibility of gene changes. The iteration is controlled by predefined parameter *iterNum*.

Algorithm 3. Genetic Algorithm for Partition: *GePA*

Input: historical workload W, schema S, iterationNumber *iterNum*
Output: partition configuration P

```
1 partitionConfig ← ∅;
2 PL = initPopulartion(W, S);
3 iter = 0;
4 while iter < iterNum do
5     rouletteSelection (PL);
6     crossOne(PL);
7     partitionConfig = variation(PL);
8     iter++;
9 return partitionConfig;
```

In Algorithm 3, new individuals are generated and the population is updated through CrossOver, Selection, and Variation steps. By continuous iterating, the partition configuration with higher cost is eliminated, and finally the optimal data partition configuration is obtained. Supposing the number of individual in population is N, the time complexity of the selection phase is $O(N * o(N))$, where o(N) is the time complexity for computing individual fitness, the time complexity of the crossover phase is $O(N/2)$ and the variation phase is $O(N)$. So the average time complexity is $O(iterNum * N * o(N))$.

4 Replication-Based Partition Algorithms

Distributed data replication can avoid data transferring with the cost of extra storage [20]. Full redundancy is infeasible especially for big data and we should find an appropriate replication strategy to minimize the data transferring cost with respect to a specified storage space.

4.1 Mixed Cost Model

A good replication strategy can effectively reduce the data transferring of distributed joins and utilize less data redundancy meanwhile. We construct a new cost model by weighting the cost of both data transferring and data redundancy:

$$Cost^P_{D,C}(W) = \alpha \times Cost^P_D(W) + \beta \times Cost_C(W)\ ,\ \alpha + \beta = 1 \tag{3}$$

Where $Cost^P_{D,C}(W)$ is the cost considering both data transferring $Cost^P_D(W)$ as in Eq. 2 and data replication $Cost_C(W)$, with respective to workload W, partition strategy P and replication strategy C. α and β are weight parameters to specify the importance of data transferring and the data storage.

4.2 Greedy Algorithm Based on Data Replication

Intuitively, data that is frequently joined remotely needs to be replicated, which can be obtained in advance by analyzing workload W and schema S. Since we may have redundant storage limitations, we evaluate data replication usefulness by Eq. 4.

$$Cost_U = \frac{Cost_D}{Cost_C} \tag{4}$$

Where $Cost_U$ is per unit data transferring cost with respect to redundancy. The higher the value, the more useful the data.

We do some modification to GePA in Sect. 3.2 for considering data replication. Running Algorithm 2, it generates the partition configuration without replications. We then calculate *unit network cost* for all the data blocks involved in all the filter conditions, which are sorted in descending order. With respect to the pre-specified space for replications, we greedily select useful data for replication.

4.3 Genetic Algorithm Based on Mixed Cost Model

We improve our genetic algorithm Algorithm 3 by combining the replication strategy. We replace the fitness function by $Cost^P_{D,C}(W)$. We extend the individual chromosome to two parts which are the original *partition* feature and current *replication* feature, as shown in Fig. 6. We add all the filters in workloads to the array as the *replication* feature, which generates potential replication data candidates. If the value for the item in replication feature is set to 1, it means data for this filter condition is preferred for replicating. The main idea of CrossOver, Selection, and Variation is roughly similar to Algorithm 3 and will not be repeated here.

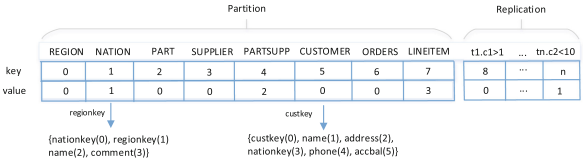


Fig. 6. Extended individual

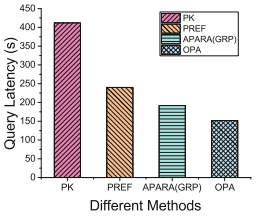


Fig. 7. Total runtime of all TPC-H queries

5 Evaluation

Apara is an assistant module for any database administrators with suggestions about partition configuration and replication strategy. We then run experiments in an open source database called GreenPlumDB [6]. We deploy GreenPlum (v.4.3) on an cluster which includes a master node and other 9 segment nodes. Each node contains two segments. For each table the data is partitioned according to the partition key by Hashing acquiescently and then divided among the segment nodes. Each machine has two 2.00 GHz 6-Core Intel(R) Xeon(R) E5-2620 processes, 120G RAM and 3.6 T local storage. The machine is running on CentOS 6.5.

We select complex workloads in TPC-H [17] for our experiments, which have 22 query templates containing more than sixty join operations. We use IPtraf [8] to collect the information of data transferring among network. Default database size is 20 GB (SF = 20) and the default replication space is set to 2GB on each node. Processing efficiency is evaluated by query latency, which is the total running time of the set of test workloads. In order to reduce cache influence, we run each set of experiments five times and take the average latency.

For comparison, we implement PK (primary key-based partition) and the idea in PREF [20] which is the latest work for locality aware query processing. In our design, we evaluate OPA, GRP (GyPA algorithm), GEP (GePA algorithm), GRPR (GyPA with replication), and GEPR(GePA with replication) in detail.

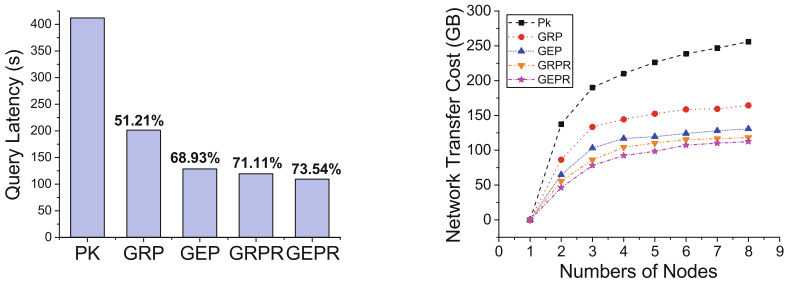
5.1 Comparison with PREF

In Fig. 7, we compare our method Apara implementing GRP with PREF [20], OPA and PK methods. We can find that Apara is better than PREF without any replication. Though OPA is the best one, its running time is unacceptable for its full traversal, which is infeasible for realistic applications when involving many tables. PREF is implemented with data replication if there is any reference relationship among tables. For Apara, if our partition uses replication, e.g., GRPR, we can get much better performance, which will be shown in the following paper. Moreover, PREF is implemented together with the modification to query processing module. So it is difficult to be applied. So next, we run experiments to compare different algorithms in Apara to show the performance.

5.2 Overview of the Performance Improvement

We compare the algorithms in Apara in Fig. 8(a). The query latency of GRP outperforms PK by 51.21%, GEP outperforms PK by 68.93%. When using replication, GRPR and GEPR can improve better, which are 71.11% and 73.54% compared to PK respectively. Figure 8(b) compares the network transfer cost with different segment nodes from 1 to 8. As we know, disk access and network transfer account for the majority time of query processing. In order to show the effect of network transfer, we set the $effective\_cache\_size$ of GreenPlum to 30GB, which are huge enough to avoid disk access. IPtraf collects network

transfer size for all the workloads. When there is only 1 nodes, no data transfer occurs. It is easy to see that with replication we can reduce network transfer cost, which are $GRPR$ and $REPR$. $Genetic-based$ methods are generally better than $Greedy-based$ methods.



(a) Query Latency for Different Methods (b) The Network Transfer Cost

Fig. 8. Overview of the performance improvement

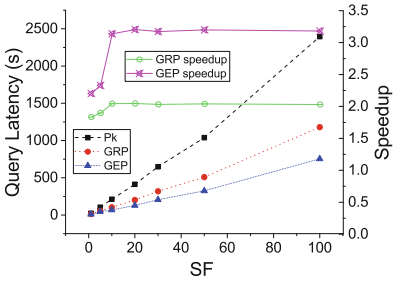


Fig. 9. Query latency with data size

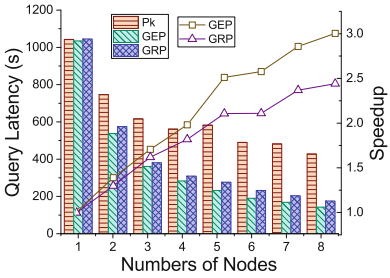


Fig. 10. Query latency with network size

Since GRPR and GEPR are based on GRP and GEP with pre-specified storage spaces, which make better performance, the scalability of both GRPR and GEPR are decided by GRP and GEP. So we only demonstrate the scalability of GRP and GEP in the following paper.

5.3 Algorithm Scalability

Scalability to Data Size: Figure 9 shows the query latency of GRP and GEP with different scale factors (SF = 1, 5, 10, 20, 30, 50, 100). We also calculate the speedup of GRP, GEP compared to PK. For the growth of data size will increase data transfer among nodes, query latency grows up for all three methods. But GEP wins the best scalability.

Speedup is increasing until SF = 10. This happens because of the bottleneck of system resources. When SF is greater than 10, CPU utility consumes almost 100% and it becomes the bottleneck. Then the latency caused by resource competition is more obvious than network transferring.

Scalability to Network Size: Figure 10 compares the query latency of GRP, GEP and PK under different Network size. When the number of node is 1, the query latency of these three partition configuration is almost the same, since there is no data transferring. As we increase the number of nodes, though data transferring among nodes will increase, parallel distributed query processing will improve performance. However GEP can effectively reduce more network cost. So GEP gets the best performance. Moreover, we can see the speedup ratio in Fig. 10 increases with the increase of nodes, so our algorithms can work well in distributed environment and have a good scalability.

Scalability to Complex Workloads: Figure 11 shows the query latency under different numbers of queries, which are selected randomly from the 22 TPC-H workloads. It compares the performance of GRP, GEP and PK. We can see that Apara's algorithms can still achieve better performance when we have larger number of workloads.

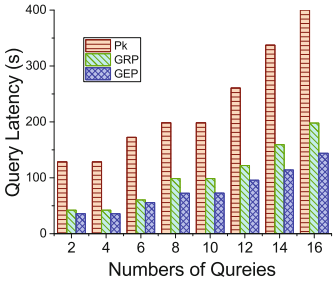


Fig. 11. Query latency with workload size

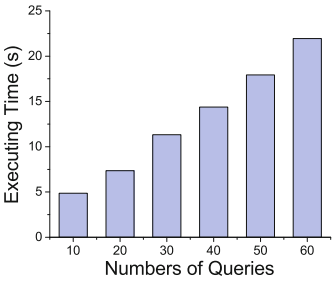


Fig. 12. Apara execution time with workload size

5.4 Apara Efficiency

Figure 12 shows the total execution time of Apara tool to get all the partition configuration suggestions for our four algorithms, which are GRP, GEP, GRPR and GEPR. Execution time of Apara approximatively linearly increases with the growth of queries selected randomly from TPC-H. Apara can get all the partition and replication configurations in a short time period, which makes it applicable for production environment.

6 Related Work

In order to avoid expensive remote join operations, data partition and replication are the feasible options for improving query performance [3,15]. Data partition can be used for both OLTP workloads [2,12,13] but also for OLAP workloads. Horizontally partition is commonly used in traditionally databases area [18]. Co-partition large tables can effectively avoid remote join operations [5]. However, in complex schemata with increasing number of tables, not all tables can be co-partitioned. Oracle [4] introduces a method REF to partition tables by reference relationships. However, this method simply partitions tables by foreign keys and does not consider the other relationships in join operators. Predicate-based reference partition [20] PREF is a partition scheme that allows to co-partition sets of tables based on given set of join predicates. However, if a referenced table in the PREF scheme contains duplicates, the referencing table will inherit those duplications as well, which may lead to a large size of redundancy. AdaptDB [9] and Amoeba [16] propose the adaptive partition mechanisms. Amoeba is designed for reducing remote disk access considering all filters conditions instead of joins. AdaptDB extends it by supporting optimization for data access in join operations, but it still has strong assumptions which is that remote disk access is the same as local disk access. There are some other work [1,11] designed based on revising query execution plans which is not applicable for production environment. Navathe [10] and HYRISE [7] are two vertical partition methods which focus on disk-based systems and main-memory resident data processing systems, respectively. Apara will consider to add vertical partition for data distribution in future.

7 Conclusion

In this paper, we present Apara, a workload-aware storage distribution guidance tool for complex OLAP workloads with the purpose to reduce the network cost and improve query performance in distributed environment. We firstly discuss the optimal partition algorithm, which is time consuming for large size of applications. Then we present two heuristic algorithms to find the potential optimal solutions. Finally, we consider to use pre-specified replication into our cost model and use it in our heuristic algorithms to further improve the performance of Apara. Our experiments show that Apara works well for complex workloads in distributed environment.

Acknowledgment. We are supported by National Key Projects (No. 2018YFB100-3404) and National Science Foundation of China (No. 61572194).

References

1. Agrawal, S., Narasayya, V., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: SIGMOD (2004)
2. Curino, C., Jones, E., et al.: Schism: a workload-driven approach to database replication and partitioning. In: VLDB (2010)
3. DeWitt, D.J., Ghandeharizadeh, S., et al.: The gamma database machine project. In: TKDE (1990)
4. Eadon, G., Chong, E.I., et al.: Supporting table partitioning by reference in oracle. In: SIGMOD (2008)
5. Fushimi, S., Kitsuregawa, M., Tanaka, H.: An overview of the system software of a parallel relational database machine grace. In: VLDB (1986)
6. GreenPlumDB. https://greenplum.org/
7. Grund, M., Krüger, J., et al.: Hyrise: a main memory hybrid storage engine. In: VLDB (2010)
8. Iptraf. http://iptraf.seul.org/
9. Lu, Y., Shanbhag, A., et al.: AdaptDB: adaptive partitioning for distributed joins. In: VLDB (2017)
10. Navathe, S., Ceri, S., et al.: Vertical partitioning algorithms for database design. In: TODS (1984)
11. Nehme, R., Bruno, N.: Automated partitioning design in parallel database systems. In: SIGMOD (2011)
12. Pavlo, A., Curino, C., Zdonik, S.: Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In: SIGMOD (2012)
13. Quamar, A., Kumar, K.A., Deshpande, A.: SWORD: scalable workload-aware data placement for transactional workloads. In: EDBT (2013)
14. Rodiger, W., Muhlbauer, T., et al.: Locality-sensitive operators for parallel main-memory database clusters. In: ICDE (2014)
15. Sacca, D., Wiederhold, G.: Database partitioning in a cluster of processors. In: TODS (1985)
16. Shanbhag, A., Jindal, A., et al.: A robust partitioning scheme for ad-hoc query workloads. In: SoCC (2017)
17. TPC-H. http://www.tpc.org/tpch/
18. Waas, F.M.: Beyond conventional data warehousing—massively parallel data processing with greenplum database. In: BIITE (2008)
19. Whitley, D.: A genetic algorithm tutorial. Stat. Comput. **4**(2), 65–85 (1994)
20. Zamanian, E., Binnig, C., Salama, A.: Locality-aware partitioning in parallel database systems. In: SIGMOD (2015)