2018 IEEE 34th International Conference on Data Engineering

A Graph-based Database Partitioning Method for Parallel OLAP Query Processing

Yoon-Min Nam, Min-Soo Kim*, Donghyoung Han
DGIST, Republic of Korea
{ronymin, mskim, icedrak}@dgist.ac.kr

***Abstract*—As the amount of data to process increases, a scalable and efficient horizontal database partitioning method becomes more important for OLAP query processing in parallel database platforms. Existing partitioning methods have a few major drawbacks such as a large amount of data redundancy and not supporting join processing without shuffle in many cases despite their large data redundancy. We elucidate the drawbacks arise from their tree-based partitioning schemes and propose a novel graph-based database partitioning method called GPT that improves query performance with lower data redundancy. Through extensive experiments using three benchmarks, we show that GPT significantly outperforms the state-of-the-art method in terms of both storage overhead and query performance.**

1 Introduction

As the amount of data to process increases, a scalable and efficient parallel database platform becomes more important. A number of parallel database platforms exist, including Apache Spark [1], Apache Impala [2], SAP HANA [3], HP Vertica [4] and Greenplum [5]. To exploit parallel data processing for OLAP queries, they typically store data blocks over a cluster of machines, execute local operations in each machine, and then repartition (shuffle) the local processing results to handle join or aggregation. Here, repartitioning is an expensive remote operation involving network communication, and its cost tends to increase as the data size or the number of machines increases [6], [7], [8].

In order to avoid expensive join operations with shuffle, a number of methods have been proposed to horizontally partition a database in an offline manner [9], [10], [11], [12]. The methods in [9], [10] co-partition only the tables containing the common join keys. However, these methods are not particularly useful for complex schema with many tables or for complex queries with join paths over multiple tables that use different join keys. The REF method [11] partitions a table R by a foreign key of R referring to another table S that is already partitioned by a primary or foreign key of S. The PREF method [12], which is the state-of-the-art method, generalizes the REF method by exploiting not only referential constraints but also join predicates (PREF-partitioning for short). PREF fully replicates manually selected small tables and partitions the remaining large tables. If query workload is available, PREF uses a workload-driven (WD) algorithm that uses the query workload to automatically find the best partitioning scheme. Otherwise, it uses a schema-driven (SD) algorithm that uses the database schema. The PREF/SD algorithm usually returns a single tree as a result, where a node indicates a table to be partitioned and an edge indicates PREF-partitioning. The root of the tree is called a *seed* table, which is hash-partitioned. Each of the seed's descendant tables is partitioned by an edge with its parent table. The PREF/WD algorithm usually returns a set of trees, i.e., a forest, as a result. PREF/WD tends to generate many trees to maximize the data-locality, where the same table might occur in multiple trees, and therefore, be duplicated many times.

Although PREF is the state-of-the-art partitioning method, it still has three major drawbacks. First, PREF/SD tends to cause a large number of tuple-level duplicates, and this tendency becomes more marked as the database schema becomes more complex. This large amount of duplicates causes the initial bulk loading of a database to be very slow. Second, PREF/WD tends to cause a large number of table-level duplicates. That is, it stores the same table many times across partitions. Third, PREF requires shuffle for query processing in many cases, despite its large data redundancy, and so, query performance tends to be degraded. Most of the drawbacks of PREF come from its tree-based partitioning scheme. In PREF, all edges in a tree or forest have a direction from source (i.e., referencing table) to destination (i.e., referenced table), which incurs so-called *cumulative redundancy* [12]. In addition, no cycles are allowed in the tree-based partitioning scheme, and so, join operations in complex queries cannot be processed without shuffle in many cases, but must be processed with shuffle. We present the above drawbacks in detail in Section 2.

To solve the above problems, we propose a novel graph-based database partitioning method called GPT. Intuitively, the GPT method determines an *undirected multigraph* from a schema graph or workload graph as its partitioning scheme. In the undirected multigraph, a vertex represents a table to be partitioned, and an edge represents a co-partitioning relationship. Since the partitioning scheme is a single graph where each table occurs only once, there are no table-level duplicates. For co-partitioning between two tables, we propose the *hash-based multi-column (HMC)* partitioning method. It is a kind of hash-based partitioning method that has no parent-child dependencies among tables. Due to no dependency among tables, it does not incur cumulative redundancy. Consequently, it results in far fewer tuple-level duplicates.

The GPT method determines the undirected multigraph so as to contain many triangles of vertices (tables). Therefore, most join operations involving the tables in these triangles can to be processed without network communication. Here, GPT determines the partitioning scheme so that these triangles have common shared vertices called *hub* tables, which improve the query performance while using less storage space. GPT also determines the partitioning scheme in a cost-based manner by considering the trade-off between the benefit of

* Author to whom correspondence should be addressed.

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00096

query processing without shuffle and the penalty of storage overhead (i.e., the number of tuple duplicates). Under this partitioning scheme, even the complex queries in the TPC-DS benchmark can be processed in a single MapReduce round in many cases.

The main contributions of this paper are as follows:

- We propose a novel and general database partitioning method for OLAP queries on parallel database systems called GPT that improves query performance with lower data redundancy. It determines an undirected multigraph as a partitioning scheme by considering both the penalty of storage overhead and the benefit of query processing without shuffle.
- We propose the hash-based multi-column (HMC) partitioning method for an edge of the undirected multigraph that has no cumulative redundancy and faster initial bulk loading.
- We propose a method for eliminating tuple duplicates that exist in the partitioned database during query processing.
- We have shown that GPT significantly outperforms the state-of-the-art method in terms of both storage overhead and query performance using the TPC-DS benchmark.

The rest of this paper is organized follows. In Section 2, we present our motivation. In Section 3, we propose the GPT method. Section 4 presents the HMC partitioning methods and the basic query processing method for GPT. Section 5 presents the experimental results, and Section 6 discusses related work. Finally, Section 7 concludes this paper.

2 Motivation

In this section, we explain our motivation and discuss the drawbacks of PREF using the TPC-DS benchmark. We first briefly introduce the PREF method. Second, we show that two kinds of duplicates, tuple-level duplicates and table-level duplicates, exist in a database partitioned by PREF in Sections 2.2 and 2.3. Third, we show that due to its tree-based partitioning schemes, query processing on top of a database partitioned by PREF tends to require shuffles during many join operations.

2.1 PREF method

PREF/SD takes a schema as an input graph and generates a set of maximum spanning trees by considering each node in the graph as a root node. That is, when the input graph has m nodes (tables), PREF/SD generates m trees. Among the m trees, PREF/SD chooses the tree that has the minimum data redundancy.

PREF/WD follows two steps. In the first step, it takes a query as an input graph and finds the best maximum spanning tree for the query as in PREF/SD. PREF/WD performs this step for every query. In the second step, it merges similar trees that have common nodes and edges into a single tree with the goal of reducing the overall data redundancy. As a result, it becomes to find several trees, i.e., a forest, where the same table might occur in multiple trees.

2.2 Tuple-level Duplicates

Figure 1 shows the schema-driven partitioning schemes determined by PREF and GPT for TPC-DS, where each box indicates a partitioned table. The tables of TPC-DS not shown in the partitioning schemes are replicated across partitions. For simplicity, we use only the abbreviations of the table names in this paper. In addition, we omit the names of partitioning columns for the tables in Figure 1. In Figure 1, the partitioning scheme determined by PREF/SD is a tree, whereas the partitioning scheme determined by GPT/SD is a graph. We note that Figure 1(b) depicts a simplified form of the actual partitioning scheme graph, which will be shown in Section 3.5.

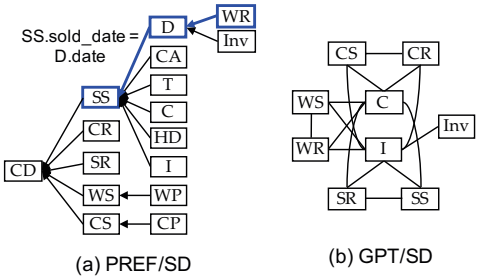

Fig. 1. Schema-driven partitioning schemes of PREF and GPT.

The tree-based partitioning scheme of PREF/SD tends to cause a large amount of duplicate tuples. For instance, in Figure 1(a), the seed table CD is hash-partitioned, and its child table SS is partitioned on a key from table CD. Then, table D is partitioned on the foreign key sold_date of table SS. This can cause a number of duplicated tuples of table D in every partition since SS.sold_date is a foreign key. Moreover, two child fact tables of D, i.e., WR and Inv, are almost fully duplicated in every partition. That is, if a referenced table (i.e., D) contains duplicates, the referencing tables (i.e., WR and Inv) also inherit those duplicates. This phenomenon is called cumulative redundancy [12]. In general, cumulative redundancy becomes more serious as the database schema becomes more complex, and the tree becomes deeper.

2.3 Table-level Duplicates

Figures 2(b) and (c) show the workload-driven partitioning schemes determined for TPC-DS by GPT and PREF, respectively. The partitioning scheme determined by PREF/WD is a forest, whereas that by GPT/WD is still a graph. The actual partitioning scheme by PREF/WD consists of seven trees, and we show only three of these trees in Figure 2(c).

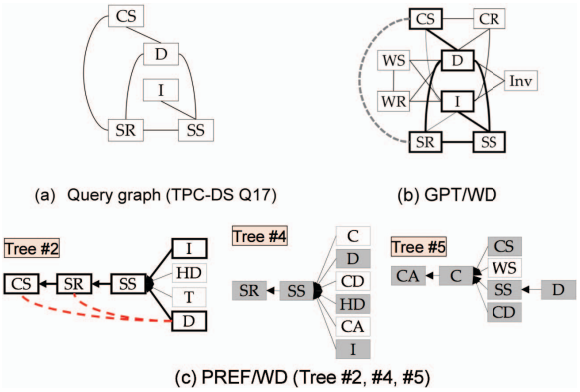

Fig. 2. Workload-driven partitioning schemes of PREF and GPT.

The partitioning scheme of PREF/WD has fewer tuple-level duplicates than does that of PREF/SD due to its lower depth

of each tree in the forest. Instead, however, it includes a large number of table-level duplicates. For instance, in Figure 2(c), the large fact table SS appears repeatedly in Tree#2, Tree#4, and Tree#5. Here, we note that the table SS is partitioned differently in each tree since the paths from the root table in each tree are different. The tables shown in gray indicate table duplicates. In general, as the query workload increases, PREF/WD determines more trees. Therefore, the number of table duplicates also increases.

In contrast, as shown in Figure 2(b), GPT/WD has no such table-level duplicates since its partitioning scheme is a single graph. The graph determined by GPT/WD is usually similar to that constructed by GPT/SD. We will explain how to determine such a graph and how to partition a table with considering its many adjacent tables in the graph in Section 3.

2.4 Join Operations with repartitioning

In spite of its large data redundancy, the tree-based partitioning schemes of PREF require shuffle during join operation in many cases, which can degrade query performance. A complex analytic query may contain cycles in the corresponding query join graph, and repartition operations are unavoidable when processing such cycles using PREF since it does not allow cycles in its partitioning scheme.

For example, Figure 2(a) shows the query join graph for the TPC-DS Q17 query, which involves five partitioned tables and nine join conditions. Each edge in the graph indicates one or more join conditions between the corresponding two end tables. When executing the Q17 query on top of the database partitioned by PREF/WD in Figure 2(c), Tree#2 is used (details in [12]). In Tree#2, the tables and join operations used for Q17 are drawn using thick boxes and thick lines, respectively. Some join operations such as SS-I and SS-D can be processed without shuffle since the database is already partitioned appropriately. However, when projecting the query join graph of Q17 onto Tree#2, two dotted red edges, CS-D and SR-D, do not exist in Tree#2, which means those two edges must be processed by join operations with shuffle.

In contrary, all the join operations of Q17 can be processed without shuffle on the database partitioned by GPT/WD. When projecting the query join graph of Q17 onto the partitioning scheme by GPT/WD in Figure 2(b), the query join graph becomes a subgraph of GPT/WD, and so, all joins can be processed in a single MapReduce round. As shown in Figure 5(b) in more detail, three tables CS, SR and SS used in Q17 are partitioned by their $date$ and $item$ columns due to two "hub" tables D and I in GPT/WD. Here, the pair of tables CS and SR have a common partition column, $item$, although we do not co-partition those tables explicitly. GPT performs hash-based partitioning on the partition column(s) of each table, and so, the tables CS and SR are co-partitioned implicitly. As a result, the join condition $CS.item = SR.item$ in Q17 can be processed without shuffle. We call this type of edge between CS and SR an *indirect join edge*. In fact, there are many other indirect join edges in Figure 2(b), but we omit them for simplicity. Our GPT method determines a partitioning scheme by considering such indirect join edges, which we will explain in Section 3. The method of generating and optimizing a query plan is beyond the scope of this paper. Instead, we present a basic query processing method for the database partitioned by GPT in Section 4.

3 GPT METHOD

In this section, we propose our graph-based database partitioning (GPT) method. GPT determines an undirected multigraph as a partitioning scheme for a given schema or workload graph. Section 3.1 introduces input join graph and output partitioning scheme. Section 3.2 presents the problem definition, Section 3.3 explains the triangles and hubs in the partitioning scheme, Section 3.4 proposes the partitioning algorithm, and Section 3.5 shows a case study using TPC-DS. We summarize the symbols used in the paper in Table 1.

TABLE I
LIST OF SYMBOLS.

Symbol	Meaning
$C(T)$	a set of partitioning columns for a table, T
$P(T)$	the horizontally partitioned table for table T
$T[i]$	i-th column of T ($i \in \mathbb{Z}^+$)
$\|\|T\|\|$	the size of T (in bytes)
$\|\|P(T)\|\|$	the size of the partitioned table $P(T)$ (in bytes)
N	the number of horizontal partitions

3.1 Join Graph and Partitioned Graph

We construct an input graph from a database schema or query workload. We simply call it as *join graph* and define it in Definition 1.

Definition 1: **(Join graph)** A join graph G=(V, E, $l(e \in E)$, $w(e \in E)$) is an undirected and weighted multigraph. A vertex $v \in V$ denotes a table. An edge $e \in E$ denotes a (potential) join relationship between two tables R and S, especially between $R[i]$ and $S[j]$, where $i \in R$, and $j \in S$. The labeling function $l(e)$ returns the equi-join predicates for edge e, i.e., $l(e) = (R[i], S[j])$. The weight function $w(e)$ returns the join frequency of the edge e.

A join graph is constructed using either a schema-driven approach or a workload-driven approach. The schema-driven (SD) approach generates a join graph $G_S = (V_S, E_S)$ based on the database schema S. The set of tables in S becomes V_S, and the set of referential constraints in S becomes E_S, which are considered as potential equi-join operations. The weight function $w(e)$ of G_S returns 1. The workload-driven (WD) approach generates a join graph $G_W = (V_W, E_W)$ based on the query workload W. The set of tables appeared in W becomes V_W, and the set of equi-join predicates in W becomes E_W. The weight function $w(e)$ of G_W returns the number of occurrences of the join predicate $l(e)$ in the workload W, i.e., the join frequency of e in W.

We denote the resulting partitioning scheme as PG, which is a subgraph of the input join graph G (i.e., $PG \subseteq G$). To determine PG, we start from an empty PG s.t. $PG.V = \emptyset$ and $PG.E = \emptyset$. We regard adding a vertex $v \in G.V$ to PG (i.e., $v \in PG.V$) as a horizontal partitioning of table v and regard not adding a vertex $v \in G.V$ to PG (i.e., $v \notin PG.V$) as replicating table v across machines. In addition, we regard adding an edge $e \in G.E$ in PG (i.e., $e \in PG.E$) as co-partitioning two end tables of e according to $l(e)$. We need to decide whether to add each v to PG or not, and also whether to add each e to PG or not.

As described above, we categorize the vertices (i.e., tables) of G into two types: $Part$-tables and Rep-tables. For a $Part$-table T, we let $C(T)$ be a subset of the columns of T used for partitioning T. We note that if $e = (R[i], S[j])$ exists in PG, then $R[i] \in C(R)$, and $S[j] \in C(S)$. When a vertex (table) T is of the $Part$ type and has no edges, then it means $C(T) = \emptyset$, and we simply split T into fixed size blocks and distribute them across machines randomly. When T is of the Rep type, we do not need to choose a set of partitioning columns for T. However, when T is of the $Part$ type, we need to choose its partitioning columns carefully since it can affect both storage overhead and query performance.

3.2 Determination of Vertices

We consider a good partitioned graph can improve the query performance largely using only a small amount of additional storage space. Without loss of generality, there are two criteria for evaluating the goodness of PG: space overhead from data redundancy and query performance improvement by co-partitioning. We try to find the optimal partitioned graph PG^* that maximally satisfies these criteria under a certain cost function. However, since there are up to $2^{|V|}$ possible combinations in terms of vertices of PG and up to $2^{|E|}$ possible combinations in terms of edges of PG, finding PG^* might be computationally prohibitive. For example, in the TPC-DS benchmark, $|V|$ is greater than 20, and $|E|$ is larger than 100, and furthermore, problems in many real applications exceed the size of TPC-DS benchmark [13], [14].

To solve this problem, we use a heuristic approach consisting of the following two steps: determining the set of vertices to be added to PG, and then, determining the set of edges among those vertices to be added to PG. We present the first step in this section, and the second step in Section 3.4.

As described above, determining a vertex v as a $Part$ type means partitioning v horizontally, while determining v as a Rep type means fully replicating v across partitions. Fully replicating a table smaller than a certain fixed threshold is a widely used technique in parallel database system [8]. PREF also uses a fixed threshold (e.g., 1000 tuples). However, GPT uses an adaptive threshold rather than a fixed one, where the decision to partition or replicate v is based on the sizes of v's adjacent tables.

Let $adj(T)$ be a set of adjacent tables of a table T in a join graph. We can formulate a total cost of I/O operations for an equi-join among table T and $adj(T)$ as shown in Eq.(1) for a T of the $Part$ type, or as shown in Eq.(2) for a T of the Rep type. Here, we assume that the cost of an equi-join between T and S ($S \in adj(T)$) is proportional to the sum of the sizes of T and S. At this point, we do not yet know the type of S, and so, regard it as a $Part$ type for simplicity. Since S is of a $Part$ type, no shuffle will be required for join between T and S in either equation.

$$PartCost(T) = ||P(T)|| \cdot |adj(T)| + \sum_{S \in adj(T)} ||S|| \cdot |C(S)| \quad (1)$$

$$RepCost(T) = ||T|| \cdot N \cdot |adj(T)| + \sum_{S \in adj(T)} ||S|| (|C(S)| - 1) \quad (2)$$

In Eq.(1), we assume that the partitioned table $P(T)$ is scanned $|adj(T)|$ times due to the join with its adjacent tables, and each adjacent table $S \in adj(T)$ is scanned once. The size of a partitioned table $P(S)$ can reach up to $||S|| \cdot |C(S)|$, since the table S can be co-partitioned with its adjacent tables using its $|C(S)|$ different columns, and no correlation exists among these columns. Likewise, in Eq.(2), we assume that the replicated table T over N partitions is scanned $|adj(T)|$ times. Here, we can regard the size of a partitioned table $P(S)$ as $||S|| \cdot (|C(S)| - 1)$ since there is no edge between T and S, and thus, there is one less partitioning column.

In Eqs.(1)-(2), it is difficult to know $|C(S)|$ for each S in advance. We can eliminate the term by calculating $PartCost(T) - RepCost(T)$ as in Eq.(3).

$$DiffCost(T) = (||P(T)|| - ||T|| \cdot N)|adj(T)| + \sum_{S \in adj(T)} ||S|| \quad (3)$$

If $DiffCost(T) \geq 0$ for a table T, we classify T as the Rep type. Otherwise, we classify T as the $Part$ type (i.e., $diff_{PR}(T) < 0$). Intuitively, under a fixed N, when table T is relatively small, and has a large number of adjacent tables whose sizes are relatively large, then $DiffCost(T)$ tends to be larger than zero. In this case, T is classified as the Rep type.

3.3 Triangles and Hubs

In real graphs, a hub vertex is one connected to many other vertices. Likewise, join graphs of real databases can include hub tables that are connected with many other tables [14]. We denote these tables as Hub-tables. We have observed that there are a lot of *triangles* of tables that share Hub-tables as common vertices in the join graphs. These triangles provide many opportunities to improve query performance via join without shuffle. For instance, Figure 2(b) shows three explicit triangles of tables, (CS, CR, D), (SR, SS, D), and (WS, WR, D), that share a hub table D. Many more implicit triangles exist due to indirect join edges, but we omit them.

We explain the concepts of *triangle edges*, *indirect join edge*, and Hub-table using Figure 3, where, for simplicity, we assume each of the five tables R, S, T, X, and Y has only a single column. In Figure 3, we regard the triangle edges $triEdges(e, T[k])$ for a given edge e and a vertex $T[k]$ as the two edges $(R[i], T[k])$ and $(S[j], T[k])$. Here, $triEdges$ $(e, T[k])$ and e form a triangle together in the join graph G. The table T might have additional columns that satisfy the above condition, and thus form multiple triangles together with e. We denote a set of those $\{k\}$ columns of T as $triCols(e, T)$. In Figure 3, we regard the indirect join edge $(X[m], Y[n])$ as one that does not exist in G, but forms a triangle together with two edges e_1 and e_2 via $T[k]$. Then, we informally define a Hub-table as a table that forms one or more triangles $\triangle(a, b, c)$ together with either an actual edge $e \in E$ or an indirect join edge, similar to T in the figure.

We note that an edge such as $e = (R[i], S[j])$ might have multiple Hub-tables that form triangles with it. We denote

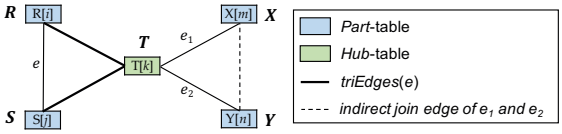

Fig. 3. Examples of triangle edges, an indirect join edge, and a Hub-table ($\{e, e_1, e_2, triEdges(e, T[k])\} \subset E$).

these as $hub(e)$ as follows:

$$hub(e) = \{T_i \mid \exists k \in T_i : \triangle(e, triEdges(e, T_i[k]))\} \quad (4)$$

In addition, we denote the set of all triangle edges that share the edge e as $triEdges(e)$:

$$triEdges(e) = \bigcup_{T_i \in hub(e), k \in triCols(e,T_i)} triEdges(e, T_i[k]) \quad (5)$$

A Hub-table T can improve query performance through horizontal partitioning T on the column $T[k]$ in many cases. Thus, we change the type of a Hub-table from the Rep type to the $Part$ type. In detail, adding an actual edge e=$(R[i], S[j])$ to PG, i.e., co-partitioning on e, allows three joins, i.e., $(R[i], S[j])$ $(R[i], T[k])$ and $(S[j], T[k])$, to be processed without shuffle. Partitioning T on $T[k]$ is also effective even when e is an indirect join edge. In Figure 3, co-partitioning on $(R[i], T[k])$ and $(S[j], T[k])$ allows processing the join operation $e = (R[i], S[j])$ without shuffle and without explicit co-partitioning on e. This approach can be particularly useful for a SD join graph, where e does not appear in a database schema, but appears in the query workload. Therefore, in general, when a Hub-table T has a higher degree, i.e., is shared among more triangles, partitioning T can further improve query performance.

3.4 Determination of Edges

Now, we discuss how to determine the set of edges among the vertices to be added to PG. Since determining the optimal set of edges for PG is still too difficult, we set a limit on the number of partitioning columns for each vertex, instead of allowing an arbitrary number of partitioning columns to be used. We denote the limit on the number of partitioning columns as κ.

It is a user-defined parameter that can control the space overhead of PG as a single knob. Since κ limits the number of partitioning columns for each table, the maximum size of an entire partitioned database is approximately proportional to κ. As we use a higher κ, the size of the partitioned database increases, and at the same time, the opportunity of join operation without shuffle also increases.

Given κ, i.e., the space overhead parameter, we can improve the query performance by choosing a set of good edges to be added to PG. To evaluate the goodness of PG, we use the concept of the *benefit* of choosing the set of edges $PG.E$. We denote the cost function for the concept by $benefit(PG.E)$, which means the sum of the amount of disk I/O for processing a join without shuffle under PG. Without loss of generality, we can say a PG that has a bigger $benefit(PG.E)$ value is a better partitioning scheme. Thus, our goal is to find the optimal partitioned graph PG^* that maximizes the benefit. We let a set of all possible PGs for a given κ be $\mathbb{PG}_\kappa$. Then, we can define our problem as shown in Problem Definition 2.

Definition 2: **(Problem Definition)** Given a database $\mathbb{D}$ and a join graph $G = (V, E, l, w)$, the problem is finding the optimal partitioned graph PG^* such that

$$PG^* = \arg\max_{PG_i \in \mathbb{PG}_\kappa} \{benefit(PG_i.E)\}. \quad (6)$$

The purpose of our GPT method is to find a near optimal partitioned graph PG ($\approx$ PG^*) that can improve the query performance largely using only a reasonable amount of additional storage compared with the original unpartitioned database. In particular, we use a bottom-up approach that adds an edge one-by-one to the initial no-edge PG. In addition, we should consider adjusting the types of some Hub-vertices of PG from the Rep type to the $Part$ type, as explained in Section 3.3. The GPT method both determines the edges and adjusts the vertices for PG in an intertwined manner.

In general, adding an edge e to PG increases the storage overhead due to tuple duplicates from co-partitioning. To measure the storage overhead, we adopt the definition of *data redundancy*(DR) for database $\mathbb{D}$ in Eq.(7) [12]. A zero DR value means that $||P(\mathbb{D})||$ is equal to $||\mathbb{D}||$, i.e., no additional storage overhead occurs from horizontal partitioning.

$$DR(\mathbb{D}) = \frac{||P(\mathbb{D})||}{||\mathbb{D}||} - 1 = \frac{\sum_{T_i \in \mathbb{D}} ||P(T_i)||}{\sum_{T_i \in \mathbb{D}} ||T_i||} - 1 \quad (7)$$

Now, we discuss the $benefit(e)$ in Eq. 6, i.e., the benefit when adding an edge e to PG, which means the amount of disk I/O required to process a join *without shuffle* thanks to $e \in PG$. We define the sum of the sizes of two end tables of e as $||e||$. The benefit increases proportionally to $w(e)$ as well as to $||e||$, where $w(e) = 1$ for a schema-driven join graph. Let us consider the case of adding the first single edge e to the no-edge PG. This addition does not increase DR since both tables are hash-partitioned on their single column, and we can say that the benefit of adding e is $w(e) \times ||e||$. Here, there is no penalty in terms of storage overhead. The GPT method chooses such edges with high priority.

To evaluate the benefit of each edge in a join graph, we categorize the edges into three types: *intra edges* among $Part$-tables, *inter edges* between $Part$-table and Rep-table, and the *indirect join edges* defined in Section 3.3. We denote three kinds of edges as E_a, E_r, and E_t, respectively. We do not consider the edges between two Rep-tables since Rep-tables already have edges with all other tables implicitly. Below, we present the benefit of initially adding an intra edge $e \in E_a$ in Eq.(8), an inter edge $e \in E_r$ in Eq.(9), and an indirect join edge $e \in E_t$ in Eq.(10).

$$benefit(e \in E_a) = ||e|| \cdot w(e) + \sum_{e' \in triEdges(e)} (||e'|| \cdot w(e')) \quad (8)$$

$$benefit(e \in E_r) = ||e|| \cdot w(e) \quad (9)$$

$$benefit(e \in E_t) = \sum_{e' \in triEdges(e)} (||e'|| \cdot w(e')) \quad (10)$$

In Eq.(8), adding $e \in E_a$ allows three join operations corresponding to e and $e' \in triEdges(e)$ to be processed without shuffle by changing $hub(e)$ to the $Part$ type, when e includes $hub(e)$. In Eq.(9), adding $e \in E_r$ allows only the join operation corresponding to e to be processed without shuffle. In Eq.(10), adding $e \in E_t$ allows the join operations corresponding to $e' \in triEdges(e)$ to be processed without shuffle. We note that e is not considered as a benefit in Eq.(10) since it is not an actual edge in a join graph. However, in case of SD join graph, the join operations corresponding to e can

exist in the query workload, and so, the edge e itself can be beneficial.

For example, we assume that six tables exist, R, S, T, X, Y, and Z, as shown in Figure 4. Then, in the figure, the red edges are the targets of $benefit(e_1 \in E_a)$ in Eq.(8), the orange edge is the target of $benefit(e_2 \in E_r)$ in Eq.(9), and the purple edges are the targets of $benefit(e_3 \in E_t)$ in Eq.(10).

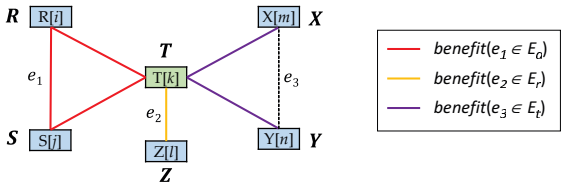

Fig. 4. Examples of three kinds of $benefit(e)$.

We present the GPT algorithm in Algorithm 1. Given a join graph G and the parameter κ, it produces a PG that can improve query performance largely while increasing DR only slightly. For brevity, we denote the $Part$-tables and Rep-tables of a join graph G as V_{Part} and V_{Rep}, respectively ($V_{Part} \cup V_{Rep} = V$).

Algorithm 1 GPT: Graph-based database ParTitioning

Input: $G = \{V, E, w, l\}$, // undirected multi-graph
κ // max # of partitioning columns per table
Variable: $benefitQ$ // max-priority queue of $\langle benefit(e), e\rangle$
Output: $PG = \{V, E\}$ // partitioned graph (subgraph of G)

1: // Step1: initialization
2: split V into V_{Part} and V_{Rep}; // according to Eq.(3)
3: add V_{Part} to $PG.V$;
4: $E_a \leftarrow \{e | e = (R.i, S.j) \in E \wedge R \neq S \wedge R \in V_{Part} \wedge S \in V_{Part}\}$;
5: $E_r \leftarrow \{e | e = (R.i, S.j) \in E \wedge R \neq S \wedge R \in V_{Part} \wedge S \in V_{Rep}\}$;
6: $E_t \leftarrow \{e | e$ is an indirect join edge $\}$;
7: // Step2: building an initial $benefitQ$
8: **for each** $e \in E_a \cup E_r \cup E_t$ **do**
9: $benefitQ.insert(\langle benefit(e), e\rangle)$;
10: **end for**
11: // Step3: adding edges and vertices to PG
12: **while** $benefitQ \neq \emptyset$ **do**
13: $\langle benefit, e\rangle \leftarrow benefitQ.extractMax()$;
14: **if** $(|C(R)| < \kappa) \wedge (|C(S)| < \kappa)$ s.t. $(R, S) \in e$ **then**
15: add $hub(e)$ to $PG.V$;
16: add e to $PG.E$;
17: add $triEdges(e)$ to $PG.E$;
18: $benefitQ.updateBenefit(adj(e))$;
19: **end if**
20: **end while**
21: **return** PG;

In the initialization step (Lines 2-6), GPT sets V_{Part} to the initial $PG.V$ and classifies E into E_a, E_r, and E_t. Then, GPT builds a max-priority queue $benefitQ$ that sorts and maintains all the edges by their $benefit(e)$. In the main step (Lines 12-20), GPT extracts the edge e with the highest benefit from $benefitQ$ and adds it to PG. Then, we check the κ constraint for the two end tables of e and add it to PG only when the constraint is satisfied. Here, if the edge e has Hub-tables as in Eq.(4), GPT also adds both $hub(e)$ and $triEdges(e)$ to PG. We note that adding $triEdges(e)$ does not increase DR at all if an edge e exists. After adding e to PG, GPT identifies the set of adjacent edges of the two end vertices of e, i.e., $adj(e)$. Then, if the sizes of two end tables increase as a result of adding e, GPT updates, especially, decreases the benefits of $adj(e)$ to reflect the loss of data redundancy. GPT repeats this main step until $benefitQ$ is empty.

3.5 A Case Study: TPC-DS Benchmark

In this section, we show the partitioning schemes determined by GPT for the TPC-DS benchmark. In Figure 5, GPT/SD and GPT/WD are quite similar with each other. PREF uses different algorithms to determine a partitioning scheme, depending on whether the input is a schema or a query workload. On the contrary, GPT uses the same algorithm in Algorithm 1 to determine the partitioning scheme, regardless of the input. The join queries in OLAP workloads are typically derived from foreign key relationships in the corresponding schema [15], and so, both GPT/SD (using a join graph from a database schema) and GPT/WD (using that from the query workload) become similar. Thus, the GPT method might be useful especially when query workload is not given, which will be shown in Section 5.3.

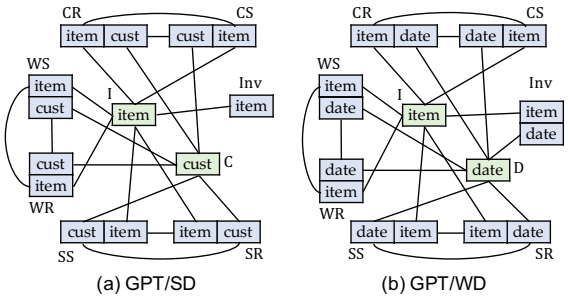

Fig. 5. Partitioning schemes of GPT for TPC-DS ($\kappa = 2$).

Both GPT/SD and GPT/WD have the same set of $Part$-tables since those $Part$-tables have large portion in both in terms of the cost model. However, they have different Hub-tables due to their different input join graphs. GPT/SD in Figure 5(a) has the tables I and C as hubs, while GPT/WD in Figure 5(b) has the tables I and D as hubs. The number of Rep-tables in GPT is greater than that of PREF. However, it is not an issue since replicating Rep-tables requires only a small amount of storage overhead (e.g., 0.2% of the whole partitioned database when the database size is 1 TB).

Both GPT/SD and GPT/WD include each table only once (i.e., no table-level duplicates), and also, have no cumulative redundancy that PREF has since they have no parent-child dependencies. Moreover, the graph-based partitioning schemes of GPT allow query processing to be performed without shuffle in most cases even for complex queries in the TPC-DS benchmark. That is partly due to a lot of indirect join edges that implicitly exist in the partitioning schemes. In Figure 5(b), GPT/WD contains 20 edges, and an additional 36 indirect join edges (a total of 56 edges). We omit the indirect join edges in the figure for simplicity. Instead, we present them in Table II. A total of 36 indirect join edges exist among the seven $Part$-tables and the two Hub-tables, D and I.

4 Query Processing

We first propose our HMC partitioning method for co-partitioning each edge in the partitioning scheme in Section 4.1. Then, we present how the scan operator eliminates

TABLE II
LIST OF INDIRECT JOIN EDGES IN GPT/WD ($\kappa = 2$).

no.	edges	no.	edges	no.	edges
1	(CR.item, Inv.item)	2	(CR.date, Inv.date)	3	(Inv.item, SR.item)
4	(CR.item, WS.item)	5	(CR.date, WS.date)	6	(Inv.item, SS.item)
7	(CR.item, SR.item)	8	(CR.date, SR.date)	9	(Inv.item, WR.item)
10	(CR.tem, SS.item)	11	(CR.date, SS.date)	12	(Inv.item, WS.item)
13	(CR.item, WR.item)	14	(CR.date, WR.date)	15	(Inv.date, SR.date)
16	(CS.item, WS.item)	17	(CS.date, WS.date)	18	(Inv.date, SS.date)
19	(CS.item, Inv.item)	20	(CS.date, Inv.date)	21	(Inv.date, WR.date)
22	(CS.item, SR.item)	23	(CS.date, SR.date)	24	(Inv.date, SS.date)
25	(CS.item, SS.item)	26	(CS.date, SS.date)	27	(SR.item, WS.item)
28	(CS.item, WR.item)	29	(CS.date, WR.date)	30	(SR.date, WR.date)
31	(SS.item, WS.item)	32	(SR.item, WR.item)	33	(SR.date, WS.date)
34	(SS.date, WR.date)	35	(SS.item, WR.item)	36	(SS.date, WR.date)

duplicates efficiently in Section 4.2 and discuss the differences between GPT and PREF in terms of data redundancy and query performance in Section 4.3.

4.1 HMC Partitioning

In this section, we present our hash-based co-partitioning method, called HMC, for the edges in PG. We define HMC partitioning to perform co-partitioning between two tables in Definition 3. We let $t.x$ be the value of column x of tuple $t \in T$.

***Definition** 3:* **(HMC partitioning)** HMC partitioning partitions a table T horizontally by hashing the column values of its partitioning column(s) $C(T)$. We denote the table partitioned by HMC partitioning as $P(T) = \bigcup_{i=1}^{P} P_i(T)$, where $P_i(T)$ is the i-th partition of $P(T)$. For a hash function $h(\cdot)$ $(1 \leq h(\cdot) \leq P)$, a tuple $t \in T$ is stored in a set of partitions $\{P_{h(t.c)}(T) | c \in C(T)\}$, where $h(t.c)$ is the hash value of the column value $t.c$ for the partitioning column $c \in C(T)$.

Since $h(\cdot)$ is applied to each column $t.c \in C(T)$ independently, a tuple $t \in T$ might be duplicated in multiple partitions when $|C(T)| > 1$. A tuple t that has $null$ values in some partitioning columns (i.e., $\exists c \in C(T) : t.c = null$), is duplicated in only the partitions $\{P_{h(t.c)}(T) | t.c \neq null \wedge c \in C(T)\}$.

Our HMC partitioning method uses bitmap information called dup in order to eliminate tuple duplicates during query processing. Tuple duplicates are common in horizontal partitioning methods [12], and thus, an efficient method for eliminating duplicates is very important. For a partition $P_i(T)$, we denote a bitmap vector of length $|C(T)|$ for a tuple $t \in P_i(T)$ as $dup(P_i(T))[t]$. In a bitmap table or a bitmap vector, 0 indicates a duplication, while 1 indicates no duplication. The content of dup bitmaps for a tuple $t \in T$ can be determined during the data loading of T. We let $C(T) = \{c_1, \ldots, c_m\}$, where $m = |C(T)|$. For a tuple $t \in T$ copied to a partition $P_i(T)$, the $dup(P_i(T))[t]$ bitmap vector is determined as $[b(1), \ldots, b(k), \ldots, b(m)]$, where $b(k) = 1$ if $h(t.c_k) = i$, but $b(k) = 0$, otherwise. We let the set of partition IDs where a tuple $t \in T$ is duplicated be $\{p_1, \ldots, p_n\}$. Then, n bitmap vectors exist for a tuple t across partitions, where $n \leq |C(T)|$. The sum of the 1s in these bitmap vectors is equal to $|C(T)|$. In a certain $dup(P_i(T))[t]$, there might be more than two 1s, when two or more column values of partitioning columns have the same hash value $h(\cdot)$. When $|C(T)| = 1$, we do not need dup bitmaps for T since only 1s exist in the bitmaps, that is, there are no duplicates.

Figure 6 shows an example of HMC partitioning for two tables R and S, where $N = 3$. The columns used for partitioning are shown in black. Table S has no dup bitmaps since it has only a single partitioning column. The first tuple of R, $(1, 3, 5)$, is copied to both partitions $P_1(R)$ and $P_3(R)$, where the bitmap vectors in $P_1(R)$ and $P_3(R)$ are $(1, 0)$ and $(0, 1)$, respectively. The second tuple $(2, 5, 7)$ is copied only to partition $P_2(R)$, where its bitmap vector is $(1, 1)$.

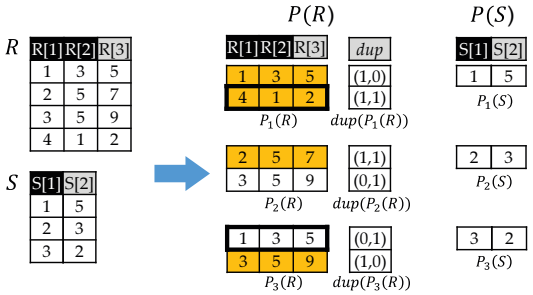

Fig. 6. Example of HMC partitioning ($N = 3$).

When storing each partition $P_i(T)$ of T, we use the concept of *subpartition* to efficiently eliminate duplicates in terms of disk I/O. We can divide $P_i(T)$ into multiple disjoint subpartitions based on its bitmap information $dup(P_i(T))$. When $|C(T)| = n$, the number of possible bitmap vectors becomes 2^n. For a given table, the number of possible dup bitmap vectors is limited since the number of possible partitioning columns is also limited. For example, if $|C(T)| = 2$, the possible bitmap vectors are $\{00, 01, 10, 11\}$ in binary strings. We create a subpartition for each distinct bitmap vector and store the tuples having the same dup bitmap vector in the same subpartition. Here, we note that GPT does not need to store dup bitmap vectors at all, since each subpartition already represents a unique bitmap vector for the tuples in the subpartition. We denote such a bitmap vector by $bitV$. As with dup, the length of $bitV$ is $|C(T)|$, and we denote $bitV$ for a subpartition $s \in P_i(T)$ as $bitV(P_i(T))[s]$.

Figure 7 shows an example of subpartitions using the same table R used in Figure 6. We assume $N = 3$ and $|C(R)| = 2$. Then, a total of $3 \times 2^2 = 12$ subpartitions are created for R. For example, a tuple $(1, 3, 5)$ is stored in subpartition 2 of $P_3(R)$ and subpartition 3 of $P_1(R)$. The tuple $(2, 5, 7)$ is stored only in subpartition 4 of $P_2(R)$ since its bitmap vector is $(1, 1)$, and its hash value is 2. Here, the $bitV$ of subpartition 4 is $(1, 1)$.

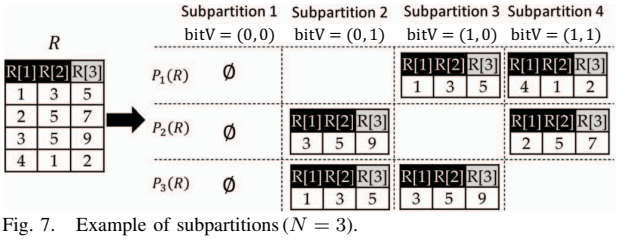

Fig. 7. Example of subpartitions ($N = 3$).

4.2 Duplicate elimination

In this section, we present how to eliminate tuple duplicates to ensure the correctness of query results. If SQL queries are executed on partitioned tables, it is essential to eliminate tuple duplicates from the query results across partitions for correctness. Existing methods [12] usually rewrite the query

plan by adding repartitioning operations, to eliminate duplicates. That approach is a kind of lazy elimination, since some tuple duplicates are carried through the pipeline of plan in each machine, and eliminated after shuffling via network communication. Carrying these unnecessary duplicates with repartitioning operations can cause extra query processing overhead.

The concept of subpartition presented in Section 4.1 allows us to eliminate duplicate tuples without carrying unnecessary duplicates and repartitioning operations in most cases. Intuitively, we can selectively access the subpartitions that are not determined as duplicates when reading a partition $P_i(T)$ from storage without false negatives and false positives. In detail, we read only the subpartitions of $P_i(T)$ corresponding to the partitioning columns of T relevant to a given query. Since the scan operator is a low-end operation, there is no need to rewrite a query plan in principle, but just need to change the scan operator so as to access to such subpartitions.

We explain how to make the scan operator be aware of the bitmap information for the following two cases : (1) scanning a table irrelevant to a join and (2) scanning a table relevant to a join.

4.2.1 Scanning a Table Irrelevant to a Join (Single-Scan Mode): If a table T does not involve join operations with other tables, we can perform duplicate elimination by checking any particular bitmap column of $bitV$ in each partition $P_i(T)$. We denote the i-th bitmap column $bitV$ as $bitV[\cdot][i]$. Although we can use any i-th bitmap column for $1 \leq i \leq C(T)$ for duplicate elimination, we just use the first bitmap column for simplicity. Then, for a subpartition $s \in P_i(T)$, $bitV[s][1] = 1$ indicates all tuples in the subpartition s are original tuples, and so, we read them from the subpartition s. In contrast, $bitV[s][1] = 0$ indicates that all tuples in the subpartition s are duplicated tuples, and so, we should not read them.

For instance, we consider scanning a partitioned table $P(R)$ in the single-scan mode in Figure 7. We assume that the first bitmap column of $bitV[s]$ $(1 \leq s \leq 4)$ is used for scanning. At the storage level, the scan operator accesses only subpartitions 3 and 4 in Figure 7. They correspond to the set of tuples in orange in Figure 6, where no tuple duplicates exist.

4.2.2 Scanning a Table Relevant to a Join (Join Scan Mode): We assume a join predicate between R and S in a query Q is $(r_1 = s_1) \wedge \cdots \wedge (r_k = s_k)$ where a set $\{r_i, \ldots, r_k\}$ is a subset of the columns of R, and a set $\{s_i, \ldots, s_k\}$ is a subset of the columns of S. Then, we let $C_Q(R)$ and $C_Q(S)$ be the sets of partitioning columns used in the join predicate for tables R and S, respectively. That is, $C_Q(R) = \{r_i, \ldots, r_k\} \cap C(R)$, and $C_Q(S) = \{s_i, \ldots, s_k\} \cap C(S)$.

Join operation without shuffle: If $C_Q(R) \cap C_Q(S) \neq \emptyset$, then it means R and S are co-partitioned with each other, and so, we can perform join operations without shuffle by using the co-partitioned column(s), i.e., $C_Q(R) \cap C_Q(S)$. The scan operator for R reads $P(R)$ in a single-scan mode, but checks the bitmap column $bitV(P_i(R))$ $[\cdot][j]$ s.t. $j \in C_Q(R) \cap C_Q(S)$, instead of $bitV(P_i(R))[\cdot][1]$. Likewise, the scan operator for S also reads $P(S)$ in a single-scan mode by checking $bitV(P_i(S))[\cdot][j]$ s.t. $j \in C_Q(R) \cap C_Q(S)$. Then, in each pair of partitions $\langle P_i(R), P_i(S) \rangle$, neither $P_i(R)$ nor $P_i(S)$ have duplicate tuples and are already co-partitioned on column j, and thus, do not require shuffle during a join without false negatives and false positives. The remaining part of the join predicate, $(r_1 = s_1) \wedge \cdots \wedge (r_k = s_k)$ except $(r_j = s_j)$ can be checked on the tuple pairs resulting from the join operation within each partition.

For instance, we consider a join between two partitioned tables $P(R)$ and $P(S)$ in Figure 6. We assume the join condition is $R[2] = S[1]$. Then, $C_Q(R) = R[2]$ and $C_Q(S) = S[1]$, and $C_Q(R) \cap C_Q(S) = R[2] = S[1]$ under the join condition. In this join scan mode, the scan operator for $P(R)$ uses the second bitmap column of $bitV(P_i(R))$, that is, accesses subpartitions 2 and 4 in Figure 7. The scan operator for $P(S)$ only scans $P_i(S)$ $(1 \leq i \leq 3)$ since there are no $bitV$ bitmaps, i.e., no subpartitions. Then, the two tuples in the bold boxes of $P(R)$ are successfully joined with the tuple of $P(S)$.

Join operation with shuffle: If $C_Q(R) \cap C_Q(S) = \emptyset$, then it means R and S are not co-partitioned with each other, and so, a repartitioning operation is unavoidable. This case is rare under GPT method since a lot of triangles and indirect join edges in GPT's partitioning scheme tend to cover a given query join graph. The scan operator for R just reads $P(R)$ in the single-scan mode described above. Likewise, the scan operator for S also reads $P(S)$ in the single-scan mode. We note that no duplicates of tuples are read from R and S. Then, the standard repartitioning operation performs join operation between R and S, and only a minimal number of tuples are shuffled as in an unpartitioned database. There is no need to eliminate duplicates during the join.

4.3 Comparison Analysis with PREF

In terms of data redundancy (DR), the DR of GPT increases proportionally with the parameter κ since each table can have up to κ partitioning columns, and the size of each table increases up to κ times under the HMC partitioning. Here, we note that the size of whole partitioned database is regardless of whether the number of partitions N increases, or the schema of database becomes more complex. The number of edges in the partitioning scheme also does not affect the DR of GPT at all, since the number of partitioning columns is still limited to κ no matter how many adjacent edges each table has. In contrast, the DR of PREF/SD increases proportionally with the number of partitions N due to its reference partitioning as shown in Figure 1(a), where a lot of D tuples are duplicated in every partition, and thus, a lot of WR tuples are also duplicated in every partition. The DR of PREF/WD tends to increase as the schema of database becomes more complex, since query trees are more diverse, and thus, more number of trees are found after merging. Each occurrence of a vertex (table) in the forest is stored independently due to its different reference partitioning. Thus, if a vertex T occurs M times in the forest, the total size of T in the partitioned database is at least M times larger than the size of the original T. If $N > \kappa$ (for PREF/SD), or the average frequency of a vertex in the forest, $avg(M)$ is larger than κ (for PREF/WD), GPT can achieve a better DR than PREF.

In terms of query performance, GPT has the following two advantages over PREF: (1) scanning smaller amount of data due to lower DR and (2) not scanning duplicated tuples due to subpartitioning. For the latter, PREF should read blocks from disks which contain some duplicated tuples. However, GPT stores each subpartition separately, each of which consists of multiple blocks, and reads only the necessary subpartitions having no duplicated tuples, which is determined by checking $bitV$, with respect to a given query. We will show the effects of (1) and (2) in Section 5.3.

5 Experimental Evaluation

In this section, we present the experimental results in three parts. First, we compare GPT with the state-of-the-art partitioning method PREF [12] to prove that GPT has both a lower data redundancy and a shorter data loading time than does PREF. Here, we evaluate GPT and PREF for both schema-driven (SD) and workload-driven (WD) approaches, while varying the number of partitions and the scale of database. Second, we compare the query performance using the database partitioned by GPT with that by PREF, in order to prove that the graph-based partitioning of GPT outperforms the tree-based partitioning of PREF. Third, we evaluate the query performance while varying the κ parameter in order to show its characteristics.

5.1 Experimental Setup

Datasets/queries: For experiments using SQL queries, we use three different benchmarks: TPC-DS [16], IMDB [17], and BioWarehouse [14]. The first benchmark, TPC-DS [16], is widely used to evaluate the performance of OLAP queries running on parallel database systems. The size of the TPC-DS database is controlled by the scale factor (SF) parameter. SF=10 generates a database of approximately 10 GB, and SF=1000 generates a database of approximately 1000 GB, which are the typical scales used in the TPC-DS benchmark. Evaluating the query performance for complex large join operations over a partitioned database can directly reveal the efficiency of a partitioning scheme. Thus, we use the TPC-DS queries that contain multiple join operations, and at least one among them uses a large fact table as an operand by following the criteria in [18], which are a total of 20 queries. The second benchmark, The Internet Movie DataBase (IMDB)[1], contains detailed information related to movies which contains a total of 21 tables and 6.4 GB data in text format[2]. The schema of IMDB is less complex than that of TPC-DS. To evaluate query performance, we use 20 queries provided by the authors in [17] which contain two large tables (cast_info and movie_info) and more than eight join conditions. The third benchmark, BioWarehouse[3], is a collection of heterogeneous bioinformatic datasets such as GenBank and NCBI Taxonomy. It contains 43 tables and 18.4 GB data. The schema of BioWarehouse is more complex than that of TPC-DS. To evaluate query performance, we use five queries provided by the authors in [14]. We implemented all the queries on top of Hadoop

[1] http://www.imdb.com
[2] ftp://ftp.fu-berlin.de/pub/misc/movies/database/
[3] http://biowarehouse.ai.sri.com

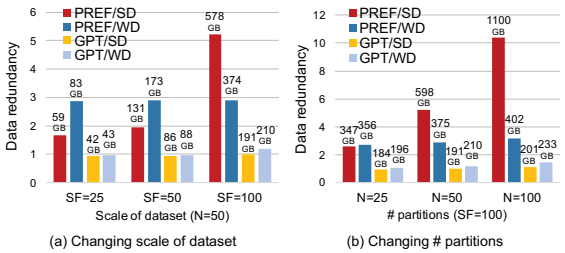

Fig. 8. Data redundancy of PREF and GPT.

and used $\kappa = 2$ for all the experiments related to GPT in default.

H/W setting: We conduct all the experiments on the same cluster of eleven machines (one master and 10 workers) in default. For the scalability experiments, we use a cluster of 21 machines (one master and 20 workers). Each machine is equipped with a six-core CPU, 32 GB memory, and two types of secondary storage (4 TB HDD and 1.2 TB PCI-E SSD). They are connected with 1 Gbps interconnection in default.

S/W setting: We use HDFS in Apache Hadoop 2.4.1 to store the datasets for all systems. For the query processing of PREF and GPT, we use the MapReduce framework in Apache Hadoop 2.4.1. We assign 6 GB memory for each map and reduce task such that up to five concurrent map/reduce tasks can be executed. To guarantee the data locality of the blocks of the same partition, we apply the custom block placement policy of HDFS as used in [19]. To obtain the partitioning schemes of PREF, we use the author's implementation[4].

5.2 Data Redundancy and Loading for TPC-DS

In this section, we evaluate the space overhead of both GPT and PREF by measuring their data redundancy (DR) with Eq.(7). Figure 8 shows the DR values of databases partitioned by GPT and PREF while changing the scale of database and the number of partitions. As Figure 8 shows, our GPT method significantly outperforms the PREF method in both the SD and WD approaches in all cases.

We note that PREF results in a low DR for a relatively simple database schema such as TPC-H as reported in [12], but it results in a very high DR for relatively complex and more realistic database schema such as TPC-DS. In particular, the data redundancy of PREF/SD increases drastically as the scale of database or the number of partitions increases due to the phenomenon of cumulative redundancy explained in Section 2. Under PREF/WD, the data redundancy does not increase since each tree is too small to incur cumulative redundancy, and only table-level duplicates exist among the multiple trees. However, it is much higher than that of GPT/WD due to its table-level duplicates. We note that the data redundancy of GPT is fairly stable regardless of both the scale of database and the number of partitions, since it mainly depends on the number of partitioning columns (κ). As a database schema becomes more complex, i.e., snowstorm schema [20], with hundreds of tables, we would expect the gap between GPT and PREF to become wider.

We also evaluate the performance of database bulk loading of GPT and PREF. Figure 9 shows the elapsed times of bulk

[4] https://code.google.com/archive/p/xdb/

loading while changing the scales of database and the number of partitions. In the figure, our GPT method significantly outperforms the PREF method in both the SD and WD approaches for all cases. These results are mainly due to data redundancy shown in Figure 8. In more detail, in Figure 9(a), the loading times of both PREF and GPT increase proportionally to the scales of database since the sizes of the partitioned databases increase. In Figure 9(b), the loading times of GPT remain fairly stable as the number of partitions increases since the sizes of the partitioned databases remain the same.

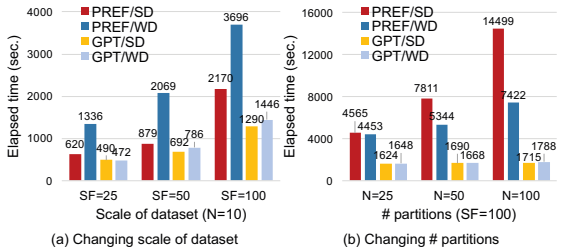

Fig. 9. Elapsed times of bulk loading of PREF and GPT.

5.3 Query Performance for TPC-DS

PREF vs. GPT: Figure 10 shows the query performance on the databases partitioned by PREF/SD, PREF/WD, GPT/SD, and GPT/WD for 20 TPC-DS queries. In this experiment, we use SF=1000 for TPC-DS and set $N = 10$. In Figure 10(a), GPT/SD significantly outperforms PREF/SD on most of the queries, although its data redundancy is much lower than that of PREF. In this figure, we note that Y-axis in this figure is log-scale. For some queries, the large sizes of PREF-partitioned tables tend to degrade the performance of query processing. For example, the two fact tables, Inv and WR, in Figure 1(a) are almost fully duplicated in every partition. Thus, the Q85 query which includes join operation between two fact tables, WS and WR, requires a large amount of I/O to scan the table WR, which severely degrades the query performance as shown in Figure 10(a). Compared to PREF/SD, GPT/SD improves the performance for the Q85 query 122 times.

In Figure 10(b), GPT/WD still outperforms PREF/WD for most of queries, despite its lower data redundancy (i.e., using smaller storage space). In Figure 10(c), GPT/WD improves the performance of PREF/WD by 48%. However, in that result, the DR of GPT/WD is only 0.92, while that of PREF/WD is 2.16. As a result, GPT/WD is 48% faster and its storage overhead is 2.35 smaller than PREF/WD. For some queries, the tree-based partitioning scheme of PREF/WD tends to degrade the performance largely due to shuffle during joins. For example, the Q17 query mentioned in Section 2 belongs to that case.

As we shown in Figure 10(c), the gap in query performance between PREF/SD and PREF/WD is huge, whereas the performance gap between GPT/SD and GPT/WD is negligible. Query performance tends to depend heavily on the partitioning scheme since the partitioning scheme represents all possible opportunities for join operations without shuffle in general. PREF/SD and PREF/WD result in quite different with each other in Figures 1(a) and 2(c), while GPT/SD and GPT/WD are very similar with each other in Figure 5. These results mean that the GPT method can be very useful especially when query workload is not available. In fact, GPT/SD outperforms PREF/SD by approximately nine times in Figure 10(c). We omit GPT/SD in the following experiments hereafter, if it is not necessary.

Scalability: Figure 10(d) shows the elapsed times for processing 20 TPC-DS queries while varying the number of machines. For this experiment, we use SF=1000 of TPC-DS and partition the database using GPT/WD. We set the number of partitions N to the number of machines. The result shows that the performance of GPT is quite scalable in terms of the number of machines. In GPT, most part of the queries are processed without shuffle, i.e., in a truly *shared-nothing* manner. In addition, the DR of the partitioned database is not affected by the number of machines, but only affected by κ. That means the amount of data to be processed on each machine decreases as the number of machines increases. As a result, the performance of GPT should be quite scalable in terms of the number of machines used.

Performance breakdown: Figure 10(e) shows the performance breakdown of our proposed method using three queries Q17, Q29, and Q85. There are three possible configurations based on two major techniques in the paper: GPT partitioning and subpartitioning. The major performance improvement comes from GPT partitioning, since it allows most of join operations to be processed without shuffle. Subpartitioning further improves the performance by 1.26-1.69 times, since it avoids scanning duplicated tuples.

5.4 Results for IMDB and BioWarehouse

Figure 11 shows the partitioning schemes of GPT/WD ($\kappa = 2$) for IMDB and BioWarehouse. As in TPC-DS, the GPT method can find a single graph that includes some hub tables for each benchmark. Compared with Figure 5(b), Figure 11(a) is small due to the simpler IMDB schema, while Figure 11(b) is large due to the more complex BioWarehouse schema.

Figure 12 shows the comparison results between GPT/WD and PREF/WD in terms of data redundancy and query performance for two benchmarks. GPT/WD outperforms PREF/WD in terms of both data redundancy and query performance. In Figure 12(a), DR of PREF/WD for IMDB is quite large due to severe table-level duplicates from lots of trees (i.e., 10) and tuple-level duplicates from many FK-FK relationships between parent and child tables. In Figure 12(b), we use the sums of elapsed times to evaluate query performance, as in Figure 12(b). We note that the gaps between PREF/WD and GPT/WD for IMDB and BioWarehouse are larger than that for TPC-DS. This is because the queries in IMDB and BioWarehouse are more complex (e.g., contain more join conditions) than those in TPC-DS.

5.5 Characteristics of GPT

In this section, we evaluate the data redundancy and query performance of GPT while varying κ. Figure 13 shows the partitioning schemes of GPT/SD when $\kappa = 1$ and $\kappa = 3$. Each table has only a single partitioning column in Figure 13(a), whereas each table has up to three partitioning columns in Figure 13(b). Only a single Hub-table appears in Figure 13(a), whereas there are three Hub-tables in Figure 13(b) due to the increased number of partitioning columns.

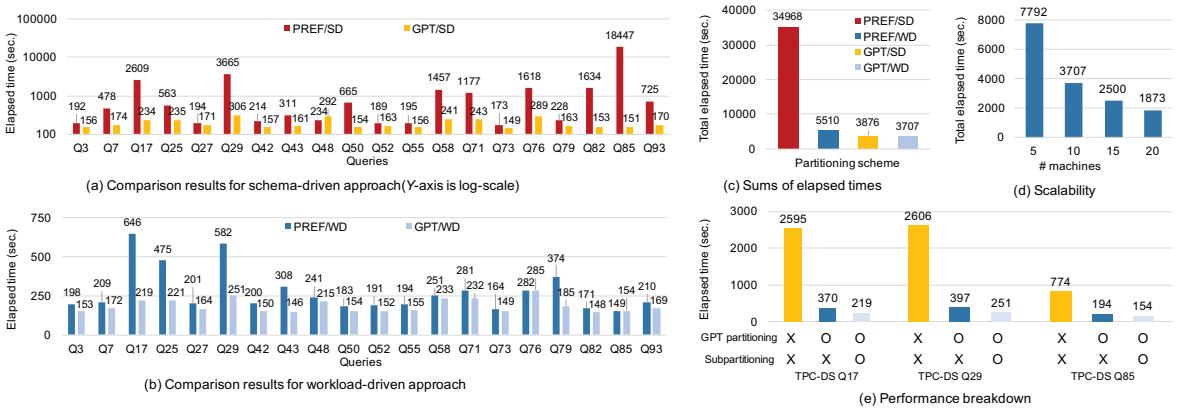

Fig. 10. (a)-(c) Query performance of PREF and GPT in elapsed times; (d) scalability of GPT; (e) performance breakdown while varying optimization techniques.

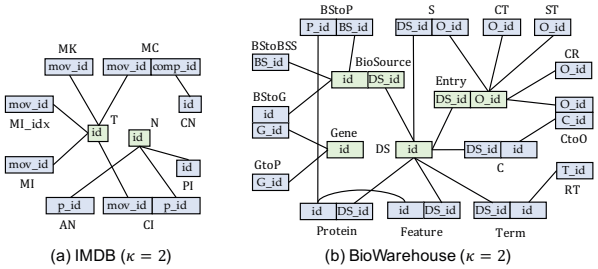

Fig. 11. Partitioning schemes of GPT/WD for IMDB and BioWarehouse.

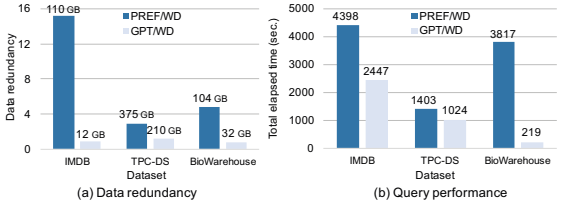

Fig. 12. Comparison using IMDB, TPC-DS (SF=100) and BioWarehouse.

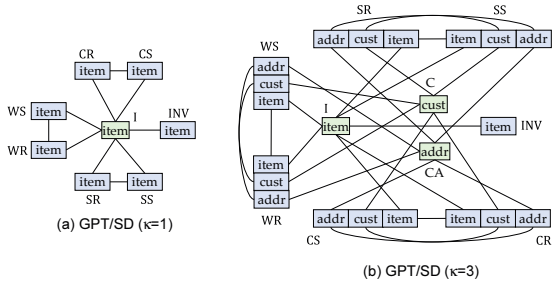

Fig. 13. Partitioning schemes of GPT/SD (varying κ).

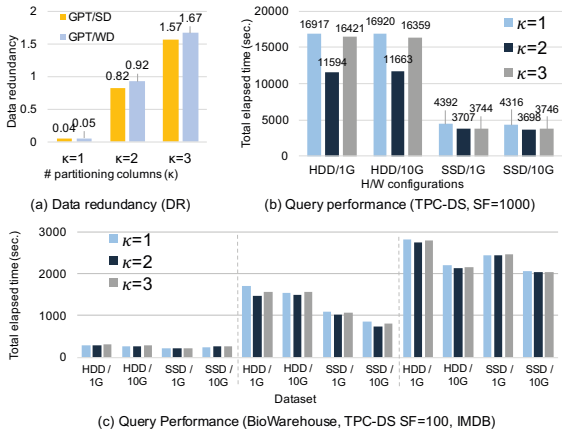

Fig. 14. Data redundancy and query performance of GPT (varying κ).

Figure 14(a) shows the data redundancy of GPT/SD and GPT/WD when using TPC-DS with SF=1000. As explained in Section 3.4, DR increases proportionally to κ. At $\kappa = 1$, there is no redundancy in $Part$-tables, but DR is non-zero due to the Rep-tables. At $\kappa = 3$, the theoretical maximum DR value is 2, but actual DR values are less than 2 due to correlations among the partitioning columns. Figure 14(b) shows the query performance of GPT/WD under a wide range of H/W settings while varying κ. Here, we use TPC-DS with SF=1000. Among three κ values, $\kappa = 2$ shows the best query performance with only a small DR (less than 1) for all H/W settings, which is coincident with the explanation in Section 3.4. A lower κ value (i.e., $\kappa = 1$) can result in worse performance due to the repartitioning overhead, while a higher κ value (i.e., $\kappa = 3$) can results in worse performance due to storage overhead. We note that using faster storage, i.e., PCI-E SSD, reduces the performance gaps among different κ values since it reduces both the storage overhead and read/write overhead of intermediate data during repartitioning. Figure 14(c) shows the query performance of GPT/WD under a wide range of H/W settings while varying κ and varying benchmark datasets. We note that the setting $\kappa = 2$ still shows the best overall performance for the different datasets.

6 Related Work

Database Partitioning Scheme for OLAP: The major performance gain in database partitioning comes from parallel query processing without repartition operations [12], [21], [22]. To remove repartition operations, it is essential to decide the appropriate partitioning columns for tables. The methods in [10], [9] co-partition the tables by their join columns, so that join operation between co-partitioned tables can be processed without shuffle. The REF method in [11] proposed reference partitioning that considers referential constraints in the table schema as partitioning predicates. Columns in referential constraint become the partitioning columns for tables, and therefore, the tables in the same referential constraint are co-partitioned. The PREF method in [12] partitions the tables based on not only referential constraints but also on join predicates. Although this approach allows the database to

be partitioned by a more number of constraints or predicates than does the REF method, it also tends to incur more data redundancy. PREF method generates tree-structured partitioning schemes that have data dependencies between the parent and child tables. Such partitioning schemes trigger two kinds of drawbacks: high data redundancy and low query performance. Our graph-based partitioning schemes solve those drawbacks of PREF method. There have been proposed skipping-oriented database partitioning methods [23], [24] which focus on scanning less data during query processing for relatively simple queries that have few join operations. AdaptDB [25] proposes an on-line partitioning method that focuses on repartitioning small portions of data continuously at runtime, but still uses tree-based partitioning schemes.

Database Partitioning Scheme for OLTP: A number of studies have been proposed to improve the performance of OLTP query processing [26], [27], [28], [29], [30]. For OLTP query processing, it is usually beneficial to use as few machines as possible for a single query since the amount of data accessed is usually quite small, and usually many queries need to be processed. Thus, parallel OLTP systems [26], [27], [28], [30] try to minimize the number of distributed transactions for given query workloads. To do that, they partition the database based on query workloads such that the overheads of query processing in the partitions is not skewed, but balanced. To partition a database, they usually create a graph with a node per tuple and edges between nodes accessed by the same transaction, and use the existing graph partitioner (e.g., METIS [31]) to split the graph into multiple balanced partitions that minimize the number of cross-partition transactions. The size of the target graph to be partitioned in these OLTP systems is quite large since it is a tuple-level graph, whereas that in our GPT is very small since it is a table-level graph.

7 Conclusions

In this paper, we have proposed a novel graph-based database partitioning method called GPT that can improve query performance largely while using only a small amount of additional storage space. Different from the state-of-the-art partitioning method PREF, the GPT method determines an undirected multigraph rather than a tree or a forest, as a partitioning scheme. GPT determines the partitioning scheme in a cost-based manner by considering the trade-off between data redundancy and the number of opportunities of join processing without shuffle. The resulting partitioning scheme contains a lot of explicit or implicit triangles of tables that can cover a query join graph in many cases, allowing a query engine to process the join query without performing repartitioning. Each edge of the undirected multigraph is assumed to be co-partitioned by the proposed HMC partitioning method with subpartitions. This approach incurs no cumulative redundancy, and results in both less overhead to eliminate duplicates and faster initial bulk loading. Through extensive experiments using three benchmarks including TPC-DS, we have shown that the database partitioned by GPT has 2.35 times smaller storage overhead than that by the state-of-the art method PREF, and at the same time, query performance using the database partitioned by GPT is 48% faster than that by PREF due to fewer join operations requiring shuffles.

Acknowledgments This work was partly supported by Institute for Information communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. R0190-15-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development), the DGIST RD Program of the Ministry of Science and ICT (17-BD-0404), and Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT and Future Planning (2017R1E1A1A01077630).

References

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *SIGMOD*, 2015.
[2] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs *et al.*, "Impala: A modern, open-source sql engine for hadoop," in *CIDR*, 2015.
[3] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: data management for modern business applications," in *SIGMOD Record*, 2012.
[4] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, "The vertica analytic database: C-store 7 years later," in *VLDB*, 2012.
[5] F. M. Waas, "Beyond conventional data warehousing: massively parallel data processing with greenplum database," in *BIRTE (Informal Proceedings)*, 2008.
[6] X. Zhang, L. Chen, and M. Wang, "Efficient multi-way theta-join processing using mapreduce," in *VLDB*, 2012.
[7] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann, "High-speed query processing over high-speed networks," in *VLDB*, 2015.
[8] S. Chu, M. Balazinska, and D. Suciu, "From theory to practice: Efficient join query evaluation in a parallel database system," in *SIGMOD*, 2015.
[9] S. Fushimi, M. Kitsuregawa, and H. Tanaka, "An overview of the system software of a parallel relational database machine grace," in *VLDB*, 1986.
[10] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, R. Rasmussen *et al.*, "The gamma database machine project," in *TKDE*, 1990.
[11] G. Eadon, E. I. Chong, S. Shankar, A. Raghavan, J. Srinivasan, and S. Das, "Supporting table partitioning by reference in oracle," in *SIGMOD*, 2008.
[12] E. Zamanian, C. Binnig, and A. Salama, "Locality-aware partitioning in parallel database systems," in *SIGMOD*, 2015.
[13] C. Loboz, S. Smyl, and S. Nath, "Datagarage: Warehousing massive performance data on commodity servers," in *VLDB*, 2010.
[14] T. J. Lee, Y. Pouliot, V. Wagner, P. Gupta, D. W. Stringer-Calvert, J. D. Tenenbaum, and P. D. Karp, "Biowarehouse: a bioinformatics database warehouse toolkit," *BMC bioinformatics*, vol. 7, no. 1, 2006.
[15] A. Weininger, "Efficient execution of joins in a star schema," in *SIGMOD*, 2002.
[16] R. O. Nambiar and M. Poess, "The making of tpc-ds," in *VLDB*, 2006.
[17] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" in *VLDB*, 2015.
[18] H. Ma, B. Shao, Y. Xiao, L. J. Chen, and H. Wang, "G-sql: fast query processing via graph exploration," in *VLDB*, 2016.
[19] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "Cohadoop: flexible data placement and its exploitation in hadoop," in *VLDB*, 2011.
[20] R. Ahmed, R. Sen, M. Poess, and S. Chakkappen, "Of snowstorms and bushy trees," in *VLDB*, 2014.
[21] J. Rao, C. Zhang, N. Megiddo, and G. Lohman, "Automating physical database design in a parallel database," in *SIGMOD*, 2002.
[22] D. DeWitt and J. Gray, "Parallel database systems: the future of high performance database systems," in *CACM*, 1992.
[23] L. Sun, M. J. Franklin, J. Wang, and E. Wu, "Skipping-oriented partitioning for columnar layouts," in *VLDB*, 2016.
[24] S. Nishimura and H. Yokota, "Quilts: Multidimensional data partitioning framework based on query-aware and skew-tolerant space-filling curves," in *SIGMOD*, 2017.
[25] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden, "Adaptdb: adaptive partitioning for distributed joins," in *VLDB*, 2017.
[26] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," in *VLDB*, 2010.
[27] A. Pavlo, C. Curino, and S. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems," in *SIGMOD*, 2012.
[28] A. Quamar, K. A. Kumar, and A. Deshpande, "Sword: scalable workload-aware data placement for transactional workloads," in *EDBT*, 2013.
[29] A. Turcu, R. Palmieri, B. Ravindran, and S. Hirve, "Automated data partitioning for highly scalable and strongly consistent transactions," in *TPDS*, 2016.
[30] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker, "Clay: fine-grained adaptive partitioning for general database schemas," in *VLDB*, 2016.
[31] 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, 1998.