Patched Multi-Key Partitioning for Robust Query Performance

Steffen Kläbe
Actian
Ilmenau, Germany
steffen.klaebe@actian.com

Kai-Uwe Sattler
TU Ilmenau
Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

Data partitioning is the key for parallel query processing in modern analytical database systems. Choosing the right partitioning key for a given dataset is a difficult task and crucial for query performance. Real world data warehouses contain a large amount of tables connected in complex schemes resulting in an overwhelming amount of partition key candidates. In this paper, we present the approach of patched multi-key partitioning, allowing to define multiple partition keys simultaneously without data replication. The key idea is to map the relational table partitioning problem to a graph partition problem in order to use existing graph partitioning algorithms to find connectivity components in the data and maintain exceptions (patches) to the partitioning separately. We show that patched multi-key partitioning offer opportunities for achieving robust query performance, i.e. reaching reasonably good performance for many queries instead of optimal performance for only a few queries.

1 INTRODUCTION

Modern data analytics include queries over large amounts of data. In order to cope with the data size, modern warehouse solutions (either on-premise or cloud) are typically designed as massively parallel processing (MPP) systems. The key for efficient query processing in this context is data partitioning, i.e. distributing chunks of data over cluster nodes in order to perform subqueries on them in parallel.

A partitioning of a relational table is defined as a disjunct split of tuples in the table and is realized using different partitioning strategies. While random partitioning or round-robin partitioning assign tuples to partitions randomly or based on their position in the table, value-based partitioning like hash partitioning or range partitioning distribute data based on the values of a chosen partition key column. The design of these partitioning strategies also aims at fulfilling the requirements of load balancing and exploiting partitioning information in query processing. While random and round-robin partitioning result in nearly perfectly balanced partition sizes, their information cannot be exploited in query optimization. On the other hand, range partitioning can be used for partition pruning in queries containing filters and hash partitioning can be exploited for partition-local execution of joins and aggregations when the partition key column matches the join or grouping key. However, load balancing hardly depends on the value distribution, leading to possibly imbalanced partitions for skewed data.

Value based partitioning strategies face a severe issue: The quality of partitioning heavily depends on the choice of the partition key to find a balanced partitioning, but also to allow partition-local execution of queries in as many cases as possible. When a user does not know which partition key to choose and the partition key consequently does not match the join or grouping column in a query, data needs to be repartitioned during query execution in order to allow parallel query execution. This repartitioning involves expensive network transfer between workers and can lead to bad query performance when choosing a wrong partitioning key, as shown for a simple aggregation query in Figure 1. Defining multiple partition keys (i.e. having a multi-key partitioning) is not possible here without replicating the data to store it in different ways.

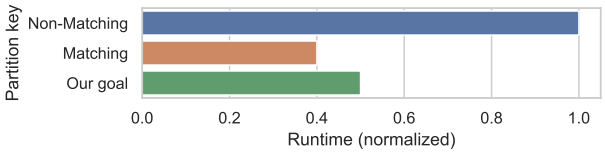

Figure 1: Groupby performance depending on matching partition key

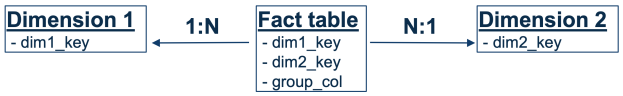

Figure 2: Minimized use case example: On which column to partition the fact table on?

Use cases for multi-key partitioning occur frequently in real world. Modern data warehouse applications contain a large number of fact and dimension tables combined in a complex schema. A minimal example for a fact table referencing two dimension tables is given in Figure 2. In addition to the foreign keys, there might be also columns in the fact table that are frequently used for aggregations, e.g. market segments or business units. So which column should be used as the partitioning key? Partitioning on one of the foreign keys results in a fast join with the respective dimension, but slower joins on other dimensions and slow aggregations on any other grouping column. Ideally, we would therefore like to partition the table on all dimension keys and expected grouping keys at the same time.

Robust query performance is an increasingly important design goal for data warehouses. It aims at achieving reasonably good (but maybe non-optimal) performance for many different workloads. Trying to find optimal query plans and chasing the last percentages of performance can lead to hitting worst-case query plans in unknown workloads due to complicated query optimizations. Robust query performance waives the aim to find optimal query plans and focuses on avoiding bad performance. For the example in Figure 1 robust performance could be slightly worse than optimal performance, but consistent for all different grouping keys. A workload-agnostic approach aims at reaching reasonable performance for an unknown workload. We do not want to optimize for a known workload that favors a certain set of e.g. grouping keys.

As partitioning has a major impact on query performance, we present a partitioning strategy that allows the definition of multiple partition keys without replicating the data in our paper. We base our approach on the idea of PatchIndexes [19, 20], which are index structures that maintain exceptions to constraints and enable query optimizations. We define the patched multi-key partitioning constraint, which captures the idea that a table is partitioned on multiple partition keys when excluding a set of exceptions for each of the columns. We show that we can find a patched partitioning of a table by mapping the table to a graph and applying existing graph partitioning algorithms. In our experiments we show the opportunities of our approach to achieve robust query performance for multiple partition keys and highlight challenges and limitations for future work. Consequently, our main contributions are as follows:

- We discuss why current partitioning schemes are not sufficient for multi-key partitioning. (Sec. 3)
- We define the patched multi-key partitioning, which allows to define multiple partitioning keys without data replication. (Sec. 4)
- We show how to find a patched multi-key partitioning using graph partitioning or iterative approaches. (Sec. 4)
- We describe how a patched multi-key partitioning can be used in query optimization (Sec. 5) and evaluate the implications for robust query performance. (Sec. 6)

2 RELATED WORK

Graph partitioning. Finding a balanced graph partitioning is known to be NP-hard [3] and typically based on graph cuts, i.e. edge-cut or vertex-cut algorithms. Conceptually, edge-cut algorithms assign vertices to partitions and remove edges from the graph that connects vertices of different partitions. Vertex-cut approaches on the other hand assign edges to partitions and remove vertices with adjacent edges of different partitions. The problem is defined and surveyed in [2, 8], discussing the challenges of distributed graph algorithms and the state-of-the-art approaches like Metis [18], JA-BE-JA [27, 28] or DFEP [14]. With Distributed Neighbour Exchange (DistributedNE) [15], a distributed graph partitioning algorithm was presented recently that achieves a proven upper bound in partition quality and outperforms the state-of-the-art approaches, scaling to trillion edge graphs and offering approaches for both edge-cut and vertex-cut. Recently, [11] showed how to incrementalize existing graph partitioning algorithms, e.g. DistributedNE, adding update support to existing graph partitionings. In our work, we do not focus on capabilities of graph partitioning approaches, but show how to map the problem of multi-dimensional relational data partitioning to graph partitioning to make use of the existing approaches.

Physical database design for OLAP. Data placement has been an investigated problem for decades, as co-locating data is crucial for query performance [34]. Besides approaches for transactional workloads [10, 25], physical database design for analytical workloads is typical workload driven, either transparent for the database system [12] or integrated into a DBMS's optimizer [23]. DBDesigner [30] recommends sort keys and column encodings besides partition keys and also takes storage footprint and fault tolerance into account. [16] presents a learned partition key advisor that learns and adapts while observing the workload and [34] presents a workload-agnostic approach besides a workload-driven one, making decisions based on the database schema and the full data. Amazon published a distribution key recommender [24] based on mapping the join history of tables to a linear program, showing that solving the problem is NP-complete. The described approaches advise optimal partition keys or partition strategies for a given workload, but are limited to single-key partitionings due to limitations of the underlying DBMS. Multi-dimensional partitioning can be achieved by holding multiple partition schemes in parallel using replication [21] or using hierarchical partitioning [29]. Orthogonally to partitioning, multi-dimensional clustering [6] or bitwise dimensional co-clustering [5] can be used to place similar tuples close to each other to reduce I/O effort of queries or efficiently support joins and aggregations by exploiting the closeness of similar tuples. In our work we envision to mitigate the limitation of DBMS's to a single partition key per table. We show that hierarchical partitioning is not suitable and achieve multi-dimensional partitioning without data replication. This opens the possibility for partition advisors to consider multiple partition keys , so queries on different partition keys can be performed without data repartitioning.

Robust Query Performance. Robust query performance is a challenge in modern database optimizer design. A topic outline and a discussion of challenges are given in [13, 32]. In general, robust query performance describes the behaviour of a system to deliver good query performance for many different workloads, but trying to avoid bad performance as much as possible. Consequently, query optimizers should not try to strive for an optimal query plan, but find a reasonable good plan that tolerates wrong decisions, like cardinality or selectivity estimation errors. Many directions exist to achieve robust performance, like runtime collection of profiling information to allow adaptive query performance (e.g. in [1, 17]), using constraint information, handling estimation errors [4, 33] or providing only a small set of physical operators to reduce the chance for bad decisions [32]. In our paper we aim at achieving robust performance on the physical data layout level by using our patched multi-key partitioning constraint to reduce repartitioning as much as possible. For all chosen partition keys we only repartition a small amount of tuples in order to achieve reasonable good performance, but never bad performance caused by a full table repartitioning. Our query optimization techniques are based on existing operators, not increasing the search space of physical operator plans as a result.

PatchIndex Overview PatchIndexes [19, 20] are generic index structures that allow the definition of approximate constraints. As typical database constraints like uniqueness can only be defined on columns that fulfill the constraint for all tuples, the useful information that all but a few tuples match a constraint can't be expressed. PatchIndexes maintain exceptions to certain constraints as sets of patches as defined in the following and make this information usable during query optimization and query execution.

Definition 2.1. **Set of patches**
Let relation R be a set of tuples t, $id(t) \in \mathbb{N}$ the tuple identifier of t and $cols(R)$ the set of columns of R. For a column C we define a set of patches $P_C \subseteq \{id(t) \mid t \in R\}$. Based on this, we define $R_{P_C} = \{t \in R \mid id(t) \in P_C\}$ as the set of tuples of R whose tuple identifiers are in P_C and $R_{\setminus P_C} = \{t \in R \mid id(t) \notin P_C\}$ as the set of tuples of R whose tuple identifiers are not in P_C.

PatchIndexes maintain sets of patches in a sharded bitmap data structure, allowing fast sequential access over an iterator for the index scan while also offering efficient insert, update

and delete operations. The PatchIndex scan is designed to split tuples on-the-fly into tuples matching a constraint and tuples not matching the constraint. It is realized using filter operators with modes *use_patches* and *exclude_patches*, which iterate over the bitmap to make the splitting decision. Consequently, both dataflows can be optimized separately, typically by dropping expensive operations on the tuples matching the constraint. We showed that "nearly unique columns" and "nearly sorted columns" can benefit from PatchIndexes by avoiding expensive distinct aggregations or sort operators [19].

PatchIndexes are generically extendable for different constraints by implementing an interface for initial creation, i.e. finding an initial set of patches, update support, i.e. maintaining the set of patches under updates, and an optimizer rule to apply PatchIndex-based query transformations. In this paper, we add the new constraint of a patched multi-key partitioning to PatchIndexes. The PatchIndex approach thereby is the key tool to enable patched multi-key partitioning, i.e. maintaining elements that do not match the partitioning criteria. Conceptually, we map relational data to graphs in order to apply existing graph partitioning algorithms and PatchIndexes will maintain graph elements that are removed by them.

3 MULTI-KEY PARTITIONING

We propose definitions for a multi-key partitioning function and a balanced partitioning and discuss drawbacks of naive ways of achieving multi-key partitioning using hash partitioning. We focus on multi-key partitionings for two keys. However, all statements can be extended to more partition keys in an obvious way.

3.1 Definitions

Definition 3.1. **Multi-key partitioning function (2 keys)** Let relation R be a set of tuples t, $cols(R)$ be the set of columns of R, columns A,$B \in cols(R)$ be the partition keys and n be the number of partitions. Further we denote $t(X)$ as the value of column X of tuple t. We define a multi-key partitioning function for two partition keys as a function $p : t \in R, n \in \mathbb{N} \mapsto i \in \{1, \cdots, n\}$ with properties:

(MK1) $\forall t_1, t_2 \in R : t_1(A) = t_2(A) \Rightarrow p(t_1, n) = p(t_2, n)$
(MK2) $\forall t_1, t_2 \in R : t_1(B) = t_2(B) \Rightarrow p(t_1, n) = p(t_2, n)$

We call **(MK1)** *Partition locality for A* and **(MK2)** *Partition locality for B*, which intuitively means that all tuples sharing the same attribute value in one of the partition keys are assigned to the same partition. The partition locality is required for partition-local execution of joins and aggregations. Only if tuples holding the same column value for a column X are assigned to the same partition, we can ensure that all matching join partners in joins or all group members in aggregations can be found in the same partition when joining or grouping on that column X. Consequently, expensive data repartitioning is not needed in this case. Combining **(MK1)** and **(MK2)** means that we aim at enabling this partition-local execution for all subkeys of a combined partition key, i.e. we want partition-local execution on both columns A and B separately if the table was partitioned on (A, B), independent on the column values of the respective other key. Please note that as a result of providing partition locality for each single-column subkey of the partition key, the multi-key partition function automatically provides partition locality for all multi-key (sub)keys of the partition key, including the whole partition key itself. This

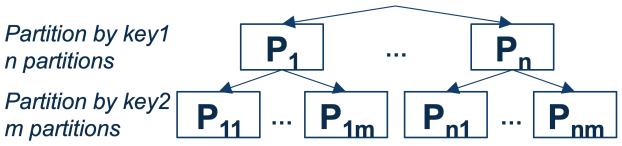

Figure 3: Hierarchical partitioning for two keys

can be easily seen by the conjunction of **(MK1)** and **(MK2)** and also holds for all subkeys of multi-key partition keys on more than two columns. Besides the requirement to enable partition-local query execution, we also require a partition function to produce a balanced partitioning to avoid load imbalance.

Definition 3.2. **Balanced partitioning** We call a partitioning of table into partitions P_i with $i \in \{1, \cdots, n\}$ balanced with an imbalance factor $\alpha = \frac{\max_{i \in \{1,\cdots,n\}} \{|P_i|\}}{\min_{k \in \{1,\cdots,n\}} \{|P_k|\}}$. Obviously, we want to have an imbalance factor α near 1 indicating a nearly balanced partitioning.

3.2 Multi-key Hash Partitioning

Hash partitioning is the most commonly used partitioning strategy, as it is easy to compute and provides partition locality in case of a single partition key. There are multiple strategies for applying hash-partitioning on multiple keys without data replication. As a first option, the partition keys can be combined to a surrogate key that is used as the input for hashing. Consequently, partitioning on columns (A, B) a tuple t with $t(A) = a$ and $t(B) = b$ is assigned to partition P_i with $i = p(a \circ b, n)$, with p being a hash-based partitioning function mapping into the value range $\{1, \cdots, n\}$ and $\circ$ being a function to combine key values to a surrogate key, e.g. concatenating, interleaving or arithmetic functions. This strategy has two drawbacks: In case of A and B not sharing the same data type, we need a mapping or conversion of the column values to a unified type the partitioning function can be applied on. More severely, we can only ensure that the hash-based partitioning function p assigns tuples to the same partition if the arguments of p are equal for both tuples. To the best of our knowledge, there is no function $\circ$ that produces the same result combining a constant a and different values for b and combining a constant b with different values for a at the same time. An exception here is the constant function producing an arbitrary constant c, which would assign all tuples to a single partition and is therefore no option. As a result, this strategy does not match the desired partition locality requirement.

As a second strategy for applying hash partitioning on multiple keys, the hash function can be applied to the keys separately and the hashed values are combined afterwards. When partitioning on columns (A, B) a tuple t with $t(A) = a$ and $t(B) = b$ is assigned to partition P_i with $i = (p(a, m) \circ p(b, m)) \bmod n$, again with p being a hash-based partitioning function mapping into the value range $\{1, \cdots, m\}$ and $\circ$ being a function to combine the hash values. Similarly to the first strategy, this strategy does not match the partition locality requirement as there is no such function $\circ$ that ensures partition locality in both arguments at the same time.

A third option for achieving multi-key partitioning without data replication is hierarchical partitioning, as shown for two partition keys in Figure 3. Data is partitioned on a single key first, and each resulting partition is further partitioned using the remaining keys. For two partition keys, a partition P_{ik} contains

all tuples that are assigned to partition i according to the first key and to partition k according to the second key. When requiring a partitioning on the first key in a query, a partition P_i consists of the union of all subpartitions $P_{ik}, k \in \{1, \cdots, m\}$, while a partitioning on the second key is constructed by the union of $P_{ki}, k \in \{1, \cdots, n\}$ for a partition P_i. This hierarchical approach has several drawbacks:

(1) Hierarchical partitioning is prone to skew. If there is a correlation between partitioning keys, it might occur that only a subset of values occur in sub-partitions after partitioning on the first key. This might lead to skewed partition sizes and can be shown using an example on the CommonGovernment table of the PublicBI benchmark [31] that we use in our evaluation in Section 6. Using hierarchical partitioning with $n = m = 4$ on *Vend_vendorname* first and on *Co_name* second leads to a partitioning where the largest partition is 3x larger than smallest partition. Using *Bureau_name* as the first partition key before *Co_name* however leads to a factor 15 between the smallest and the largest partition size. The reason for this is the column correlation. *Bureau_name* is highly correlated to *Co_name* (see Figure 9), so 99% of the distinct *Co_name* values can be found in only one of the four partitions after the first partition step. *Vend_vendorname* however does not have a strong correlation to *Co_name*, so around 80% of the unique *Co_name* values can be found in all partitions after the first partitioning step, which is more similar to a random distribution and consequently leads to more balanced partitions after the second partitioning step.

(2) Partition responsibilities change during query execution. An executor is responsible for partitions $P_{i\_}$ when requiring a partitioning on the first key, while being responsible for $P_{\_i}$ when requiring a partitioning on the second key. Nearly all partition responsibilities are moved between queries, making e.g. buffered data useless.

(3) The number of partitions increases rapidly with the number of columns and cores. The total number of partitions is $\prod_{i=1}^{num\_cols} n_i$, with n_i being the number of partitions for key column i. In order to avoid unused cores during query execution, we require $n_i \geq num\_cores; 1 \leq i \leq num\_keys$. For an example machine of 128 cores and three partitioning keys we would need at least 2^{21} partitions. This leads to an non-negligible overhead in query execution due to additional metadata and partition assignments.

(4) The order of partition columns impacts load balance.

(5) Due to changing node responsibilities for partitions, a shared-disk approach is a requirement for hierarchical partitioning.

Changing data responsibilities impacting buffering is a major drawback of hierarchical partitioning. (3) and (4) could be mitigated by applying a physical allocation strategy of the partitions to physical partitions, e.g. in [29]. (5) is a minor drawback here, as data access for all nodes on all data is typically ensured by either a cloud file system or a shared file system in an on-premise cluster setup. The access however might be non-uniform, having local and remote reads like in HDFS.

4 PATCHED MULTI-KEY PARTITIONING

In order to mitigate the drawbacks discussed in Section 3.2, we relax the multi-key partitioning definition and introduce patched multi-key partitioning, i.e. allowing exceptions to the partitioning constraint. We show how to find patched multi-key partitionings using graph partitioning algorithms or iterative algorithms.[1]

Algorithm 1: Iterative graph partition assignment

Input : Relation $R = \{t_i | 0 \leq i \leq n\}$,
Partition keys $K \subseteq cols(R)$

Output : Set of partitions $P = \{P_i | P_i \subseteq R, i \in \mathbb{N}, \bigcup P_i = R, \forall i, k \in \mathbb{N} : P_i \cap P_k = \emptyset\}$

```
P ← {}
while R \ ⋃P ≠ ∅ do
    ΔR ← R \ ⋃P
    X ← {}
    X' ← {t} for some t ∈ ΔR
    while X ≠ X' do
        X ← X'
        for k ∈ K do
            // No duplicates due to set semantics, so no
               need to adapt ΔR
            X' ← X' ∪ {t ∈ ΔR | ∃t' ∈ X' : t(k) = t'(k)}
    P ← P ∪ {X} ;
return P
```

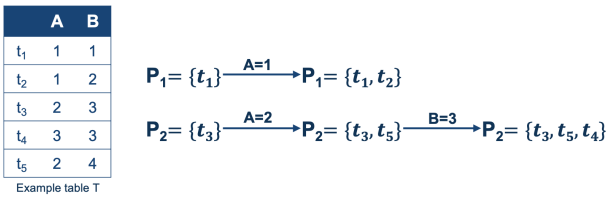

Figure 4: Sequence of the naive multi-key partitioning

4.1 Motivation

Based on Definition 3.1 there is a straight-forward and inevitable approach to find a multi-key partitioning on a given dataset. This approach is sketched in Algorithm 1 and serves as a starting point for further discussions. The main idea is to iteratively build data partitions by starting at a single tuple and adding tuples in order to match **(MK1)** and **(MK2)**, which means that in one step we add all tuples with matching values in one of the partition keys, and repeat until we do not find any tuples to add. If there are still unassigned tuples we repeat this behaviour starting with an unassigned tuple. The resulting partitions can then be used directly as table partitions or assigned to physical partitions using a physical allocation strategy as e.g. in [29]. An example sequence of this algorithm is shown in Figure 4. Starting with tuple t_1 in the first partition P_1, we add t_2 in the first step as t_2 also contains the column value $t_2(A) = 1$. The loop will now terminate after checking that no more tuples share values for columns A or B with tuples in partition P_1. As not all tuples are assigned yet, we start a second partition P_2 initially containing t_3. t_5 is added next due to a matching value in column A and t_4 is added due to matching value in column B. The final partitioning consequently consists of two partitions.

We implemented the algorithm using a recursion of SQL semi-joins and unions for initial experiments. These experiments revealed the major drawbacks of the naive approach, as it produces

[1] Available under https://github.com/dbis-ilm/Patched_Multi-key_Partitioning.

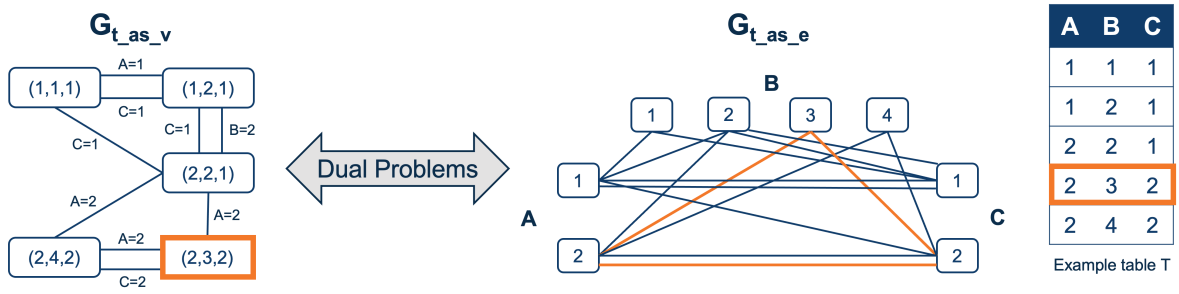

Figure 5: Graph constructions for example table with 3 partition keys

a single large partition for most datasets and partition keys. Even if it discovers multiple partitions in the data, the number of physical partitions and their size is determined by this "natural" number of partitions in the given dataset. We can allocate partitions to physical partitions to reduce the number of partitions, but we can not fill more partitions than "naturally" discovered. However, if we would agree that the value $A = 2$ is an exception and tuples holding the value $A = 2$ are not necessarily assigned to the same partition, tuple t_5 would not be assigned to P_2 and would consequently form a separate partition P_3. A query containing a grouping on column B could be executed partition-locally without the need for data repartitioning. Grouping on A also does not require to repartition all tuples, but only the tuples holding the exception value $A = 2$. This example motivates the definition of a patched multi-key partitioning.

4.2 Definition

We want to allow marking tuples as exceptions to the multi-key partitioning as defined in Definition 3.1 in order to find more balanced partitions and to be independent from the "natural" partitions of a given dataset. Keeping in mind that PatchIndexes allow the definition of approximate constraints as described in Section 2 by maintaining sets of patches, we therefore relax the definition of a multi-key partitioning as follows:

Definition 4.1. **Patched multi-key partitioning (2 keys)** Let relation R be a set of tuples t, $cols(R)$ be the set of columns of R, columns $A, B \in cols(R)$ be the partition keys and n be the number of partitions. Further we denote $t(X)$ as the value of column X of tuple t and assume the existence of a set of patches P_c for every partition key column $c \in \{A, B\}$. We define a patched multi-key partition function for two partition keys as a function $p : t \in R, n \mapsto i \in \{1, \cdots, n\}$ with the following properties:

(PMK1) $\forall t_1, t_2 \in R_{\setminus P_A} : t_1(A) = t_2(A) \Rightarrow p(t_1, n) = p(t_2, n)$
(PMK2) $\forall t_1, t_2 \in R_{\setminus P_B} : t_1(B) = t_2(B) \Rightarrow p(t_1, n) = p(t_2, n)$

Intuitively, **(PMK1)** and **(PMK2)** relax the *Partition locality* to tuples that are not included in the set of patches for the respective column. The definition also does not make any claims about the size of the set of patches, i.e. a trivial solution would be to assign all tuples to the sets of patches. The main goal is now to construct both a partition function and small sets of patches for the partition key columns, such that the defined constraints are met.

4.3 Graph Partitioning

4.3.1 Overview. The naive algorithm presented in Section 4.1 showed that the problem of finding a (patched) multi-key partitioning intuitively can be transferred to graph algorithms. The constraints that once a tuple is assigned to a partition, every other tuple sharing the same value in at least one of the partition key columns must be assigned to the same partition, are similar to following paths in a graph to discover its connected components. The "natural" partitions that Algorithm 1 discovers are thereby the connected components of a graph constructed from the tuples of the given dataset. We formally define a mapping of relations to graphs and exploit existing graph partitioning algorithms as a first approach for meeting the goal of finding sets of patches and a partitioning function such that Definition 4.1 is met. The main concept of using graph partitioning algorithms can be depicted as follows:

(1) Map the input table to a graph based on the partition keys.
(2) Apply a graph partition algorithm on the graph. This results in an assignment of vertices/edges to a graph partition.
(3) Map the graph partitions back to table partitions.
(4) Define set of patches based on overlapping vertices or edges.

By defining the mapping between graphs and tables, this approach can be generalized to arbitrary graph partitioning algorithms. In the following, we describe generally applicable mappings. Only the actual storage layout of the graph is dependent on the expected input layout of a given graph partitioning algorithm.

4.3.2 Table to Graph Mapping. Mapping a table to a graph is based on the chosen set of partitioning keys and can be done in two general ways: Modeling tuples of the table as vertices or as edges, resulting in dual representations depicted for an example table in Figure 5 and defined as follows:

Definition 4.2. **Graph Construction, Tuples as Vertices** Assume a Relation R and a set of partition keys $P \subseteq cols(R)$. We define a multigraph $G_{t\_as\_v}(R) = (V, E)$ that represents the relation R with $V = \{t | t \in R(P)\}$ and $E = \{(u, v)_k | u, v \in R, u \neq v, k \in P : u(k) = v(k)\}$.

Definition 4.3. **Graph Construction, Tuples as Edges** Assume a Relation R and a set of partition keys $P \subseteq cols(R)$. We define a multigraph $G_{t\_as\_e}(R) = (V', E')$ that represents the relation R with $V' = \{v | \exists k \in P : v \in R(k)\}$ and $E' = \{(u, v)_t | t \in R, k_1, k_2 \in P, k_1 \neq k_2 : t(k_1) = u \wedge t(k_2) = v\}$.

Intuitively, the graph $G_{t\_as\_v}(R)$ contains a vertex for each distinct combination of partition key values of R and an edge between two vertices for each shared partition key value. On the other hand, $G_{t\_as\_e}(R)$ is a multipartite graph and contains a vertex for each distinct partition key value and an edge between two vertices for each tuple in R which contains the respective partition key values. As columns might share column values leading to conflicts in vertex identifiers, we use a unique mapping between

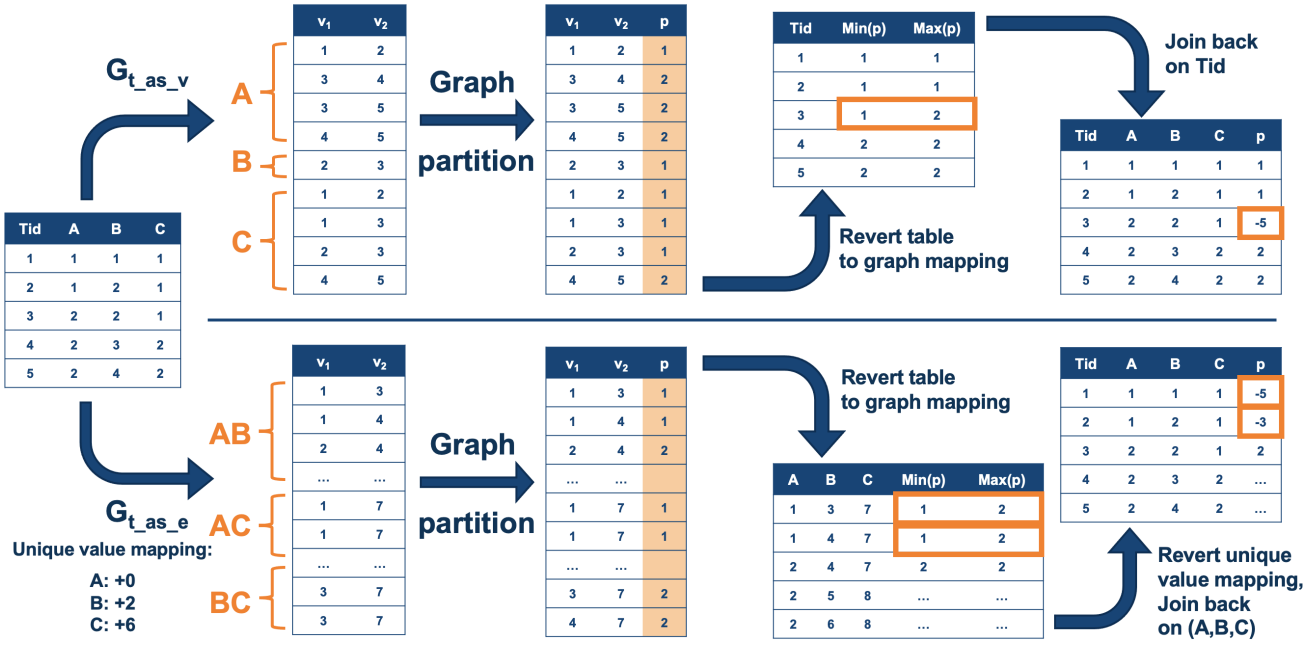

Figure 6: Graph partitioning example workflow

partition key values and vertex identifiers in the implementation. We defined both graph constructions as multigraphs, which could be replaced by constructing a weighted graph with weights indicating the number of tuples holding the pair of values in the case of $G_{t\_as\_e}$ or the number of equal partition key values in the case of $G_{t\_as\_v}$. Both approaches work similarly and the choice between them is only impacted by the expected input of the graph partitioning algorithm. Conceptually, an edge with a large weight indicates the same graph connectivity as multiple edges between vertices. In Figure 5, we used the multigraph construction, so there are two edges between the vertices $(2, 4, 2)$ and $(2, 3, 4)$ in $G_{t\_as\_v}$ or two edges between $A = 2$ and $C = 2$ in $G_{t\_as\_e}$ respectively. The resulting temporary graphs can be larger than the original table. With n being the number of tuples in the base table, d_i being the number of distinct values in column i and k being the number of partition key columns, graph $G_{t\_as\_v} = (V, E)$ has size $|V| = n$ and $0 \leq |E| \leq k \cdot \sum_{i=1}^{n-1} i$ (depending on the common partition key values of e tuples; minimum: all unique, maximum: all tuples equal) and graph $G_{t\_as\_e} = (V', E')$ has dimensions $|V'| = \sum_{i=1}^{k} d_i$ and $|E'| = n \cdot \sum_{i=1}^{k-1} i$, particularly larger than n as tuples can be represented by multiple edges for more than two partition key columns.

In our implementation we realized both constructions over SQL queries to create the graph representation from a given database table. The required graph representation is determined by the expected input of the used graph partitioning algorithm. In our case, we produce a flat file containing a list of edges. Figure 6 shows an example end-to-end workflow for the graph partitioning approach. For $G_{t\_as\_v}$ we use tuple identifiers as vertex identifiers and calculate edges by performing a self join of the table for every partition key separately and combining the results using a union. For $G_{t\_as\_e}$ we use the column values of the partition keys as vertex identifiers and apply a unique column mapping (in the example: adding the running sum of max values of other partition keys). We project all two-element subsets of the partition key, apply the unique column value mapping and union the results of every subset.

4.3.3 Graph partitioning. In the next step, we apply a graph partitioning algorithm on the constructed graph. As discussed in Section 2, there is the choice between vertex-cut and edge-cut algorithms, and both alternatives can be applied to both possible graph constructions. While vertex-cut algorithms assign edges to partitions and cut vertices from the graph, edge-cut algorithms assign vertices to partitions and cut edges from the graph. Cutting the equivalent of tuples in the graph, i.e. edge-cut in $G_{t\_as\_e}$ and vertex-cut in $G_{t\_as\_v}$, cuts whole tuples and consequently leads to a single set of patches for the whole table. Using the respective other variant leads to a single set of patches for each of the partition key columns. This offers more flexibility, so we expect smaller sets of patches when later querying columns of the partition key separately.

The remaining two options, i.e. performing an edge-cut on $G_{t\_as\_v}$ or a vertex-cut on $G_{t\_as\_v}$ also show some challenges. In $G_{t\_as\_v}$, edges represent a common column value between two tuples. Removing an edge from the graph however only has a local impact and does not impact other edges representing the same column value. As we described in Section 4.1 and will describe formally when describing query execution using PatchIndexes in Section 5.2, we need to repartition all tuples sharing a column value that was marked as an exception. Consequently, we would need to remove all edges with a common column value when removing one of them, e.g. removing all edges representing $A = 2$ in Figure 5. Conceptually, this issue is caused by the fact that a single column value is represented by multiple graph objects, i.e. multiple edges in this case. On the other hand, $G_{t\_as\_e}$ shares a similar challenge. Here, tuples are represented by multiple edges. Performing a vertex-cut on $G_{t\_as\_e}$ assigns edges to partitions and could therefore possibly assign a tuple to different partitions, which leads to conflicts when mapping the graph back to the table.

The property of finding a balanced partitioning and minimizing the number of cutted graph objects (and minimizing the sets of patches later) relies on the properties of the graph partitioning algorithm. As we focus on embedding any given graph partitioning algorithm into relational table partitioning in this paper, discussions about the quality of the partitioning produced by a choosen algorithm is out of scope of this work. As an example, we chose to use the vertex-cut algorithm Distributed Neighbor Extension [15] on both the $G_{t\_as\_v}$ and the $G_{t\_as\_e}$ construction, comparing the effects described above. DistributedNE starts with a random edge per partition, iteratively grows partitions in parallel using neighbor expansion and performs synchronisation between the expansion rounds. The algorithm finds a local optimum, but due to the random starting point selection it does not guarantee a global optimum.

4.3.4 Graph to Table Mapping. Mapping the partitioned graph back to the relational table is the last step and includes solving the two main goals stated in Section 4.2, namely to find a partition function and sets of patches for partition key columns in order to meet the constraints of a patched multi-key partitioning. After applying the graph partitioning algorithm that assigned graph elements to partitions we reverse the graph construction by adding the assigned partition to the respective tuple as an additional table column. This way, we materialize the partitioning function as a mapping directly in the table. Afterwards we repartition the table on the graph partition column using regular hash partitioning. DistributedNE performs a vertex-cut, so it assigns edges to partitions. In order to reverse the mapping described in Section 4.3.2 we create a table of schema $(vertex1, vertex2, partition\_id)$ from the partitioning result file like shown in Figure 6 in order to perform the mapping again over SQL. For $G_{t\_as\_v}$ we produce a mapping of partition identifiers to tuple identifiers by projecting $(vertex1, partition\_id)$ and $(vertex2, partition\_id)$ and union them, before grouping on the vertex column and computing the minimum and maximum partition_id. If minimum and maximum match, the tuple was not cut from the graph by the vertex cut and we can apply the partition_id. Otherwise we apply a negative partition_id and the tuple gets assigned to the sets of patches later. For $G_{t\_as\_e}$ we split the table to reconstruct the two-element partition key subset based on filter predicates (using the unique value mapping). Afterwards we join these parts to reconstruct a mapping table of schema $(partition\_key\_columns, partition\_id)$. Again, we group on the partition key columns and calculate the minimum and maximum partition_id to discover and mark conflicts like in the $G_{t\_as\_v}$ case. For both cases we need to ensure that the partition_ids calculated by the graph partitioning are also assigned to different physical table partitions, which is not necessarily ensured, i.e. it is not ensured that partition_ids between 1 and n are indeed assigned to n different physical partitions. We therefore update the partition_ids with values that are ensured to be assigned to different partitions. These can be easily calculated if the hash function used for hash partitioning is exposed over the SQL frontend of the database system. The reverted mapping tables are finally joined back with the fact table on the tuple identifier for $G_{t\_as\_v}$ or on the partition keys for $G_{t\_as\_e}$ in order to update the materialized partition_id column.

In the last step, we need to identify the sets of patches. Graph elements that have adjacent graph elements with different partitions are exceptions, which are removed from the graph in the graph partitioning. Similarly, we query the table for each partition key to identify partition key values holding more than one distinct partition_id or a negative partition_id indicating a conflict from the graph to table mapping as described in Section 5.1. These tuples are declared as exceptions and assigned to the set of patches for the respective partition key column. In the upper case of the example in Figure 6, all tuples with $A = 2$ would be included in the set of patches P_A for column A, tuples with $B = 2$ in P_B and tuples with $C = 1$ in P_C respectively, because tuples holding these values have either more than one distinct partition_id or a negative partition_id.

4.3.5 Discussion. The approach of mapping a relational table to a graph, using existing graph partitioning algorithms to partition the constructed graph and infer the table partitioning from the graph partitioning is an elegant way of combining both worlds to apply the well-known and well-proven characteristics of graph algorithms to relational tables. However, it also has some drawbacks. First, the dataset must be completely loaded before graph partitioning can be invoked. This is fundamentally different from a typical data ingestion pipeline, where tuples are iteratively loaded (in parallel), and decisions on partition assignment are made locally to a tuple instead of considering the whole table. Consequently, we need to repartition the table after it was already loaded to apply the graph partitioning. Second, data layouts in analytical database systems fundamentally differ from data layouts in graph databases, so expensive data reorganisation or data transport would be needed. In our case, we export the constructed graph from the database in order to use it as an input for DistributedNE. Although iterative extensions [11] exist that would enable iterative partition assignments during loading, they would either require reconstructing the graph when data should be appended to an existing table or the constructed graph needs to be maintained as a separate copy of the data during the table lifetime. For both reasons, the approach described above might be hardly applicable in practice, but motivates the design of a iterative patched multi-key partitioning approach.

4.4 Iterative Patched Partitioning

Using graph partitioning algorithms require the full dataset to be present and are difficult to integrate due to the different data layouts of relational databases and graph systems. However, algorithms like DistributedNE ensure minimal sets of cuts (which translates to minimal sets of patches) under given balancing constraints. In order to make the approach more usable in practice, we design an iterative patched partitioning strategy as an alternative approach in this section which is able to decide on partition assignments upon coming across a single tuple at the price of loosing the optimality of the resulting partitioning. The main idea of the approach is to iteratively build and "color" a graph. Similar to hash partitioning, we thereby make a partition assignment decision just in the moment the tuple is loaded into the table. Our example algorithm shown in Algorithm 2 is based on the $G_{t\_as\_e}$ construction and intuitively holds an assignment of values to partitions for each column of the partition key. For each tuple, we need to distinguish between three cases. If there is no partition assignment for any of the partition key values yet, we apply a partition function f on it. If there are some key values that already have an assignment (because they already occured before) and all of these partition_ids match, the tuple is assigned to this partition, matching the intuition that tuples sharing a partition key value are assigned to the same partition. If

Algorithm 2: Iterative graph partition assignment

```
   Input  : *part_key_values, num_keys
   Output : partition_id
1  lookups ← part_assignment_lookup(part_key_values);
2  if allNULLOrExceptions(lookups) then
3  |  part ← f(part_key_values) ;                    // Case 1
4  else if allNULLOrExceptionOrEqual(lookups) then
5  |  part ← firstAssignment(lookups) ;              // Case 2
6  else
7  |  global rotating_idx ;                  // Case 3: Conflict
8  |  while lookups[rotating_idx] == NULL OR
   |   lookups[rotating_idx] == EXCEPTION do
9  |  |  rotating_idx ← (rotating_idx + 1) % num_keys ;
10 |  part ← lookups[rotating_idx];
11 |  rotating_idx ← (rotating_idx + 1) % num_keys ;
12 global partition_mappings ;
13 for i = 0; i < num_keys; i = i + 1 do
14 |  if lookups[i] == NULL then
15 |  |  partition_mapping[i].insert(part_key_values[i], part)
16 |  else if lookups[i] ≠ part then
17 |  |  partition_mapping[i].update(
   |  |   part_key_values[i], EXCEPTION)
18 return part ;          // Partition_id for given tuple
```

however there are key values already assigned to a partition and these assignments do not match, we have a conflict. This tuple would conceptually connect two connectivity components in the iteratively built graph. We can't revert previous decisions to join these connectivity components and decide for one partition key to be the decisive key, so the tuple will be an exception for every other column when later discovering the sets of patches. The choice for the decisive key follows a rotating schema in order to distribute exceptions over all partition key columns. In all cases, we insert the assigned partition_id into the assignment for every column that did not have an assignment before or update to the exception marker when hitting case 3, i.e. there is a partition_id for a value which was not chosen. The exception markers are handled similarly to NULLs in case 1 and case 2, because they are not decisive.

The described algorithm obviously has some drawbacks. Due to the "local" decision in the conflict case, it is not able to find a globally optimal partitioning with a minimum set of patches. Second, it does not have any guarantees about partition balancing. The first case thereby has the most impact on balancing. We track the current sizes of each partition (not shown in Algorithm 2) and the function f assigns a tuple to the smallest partition in the in order to fill partitions as equally as possible. Third, the algorithm is sensitive to the order of tuple insertion. As a simple example depicted in Figure 7, inserting tuples that represent a path through a graph would assign all tuples to a single partition, so we rely on the assumption that we hit the first case regularly during tuple insertion to assign tuples to different partitions. Inserting the tuple with partition keys $(2, 2)$ in the upper case of Figure 7 and hitting case 3, the tuple (i.e. the edge) is assigned to the second partition according to key A (indicated by not connecting the edge), so all tuples with $B = 2$ will later belong to the set of patches P_B as they belong to different partitions. Due to the rotating index for choosing the decisive column, column B would be chosen as the decisive column when hitting case 3 for the next

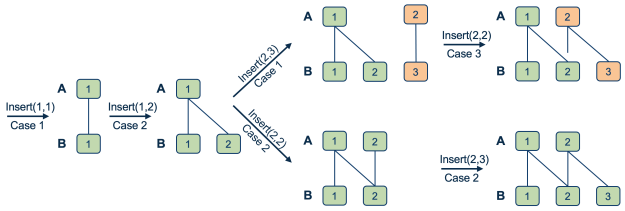

Figure 7: Order-sensitivity of iterative patched partitioning

time in order to balance patches over all partition keys. However, having a decision making process "local" to a tuple enables on-the-fly partition assignments during data loading, avoiding expensive conversions and repartitionings of the graph partitioning approach presented in Section 4.3. The algorithm is designed as a single-threaded approach. In order to be integrated into parallel loading, access to the metadata structure must be secured using locks during Algorithm 2 between reading the partition assignments and updating them. As this is the major part of the algorithm, an optimistic approach would be favourable here, i.e. performing the partition assignment lookup again before updating to ensure that no updates were performed in the meantime. In order to make it openly available, our implementation contains a standalone version of the algorithm not integrated into data loading operators inside a database kernel. The standalone version processes input files and produces an output file with an additional column for the computed partition_id.

5 QUERY PROCESSING

As described in Section 2, PatchIndexes are generic data structures that can be extended to arbitrary constraints by implementing an interface for PatchIndex creation, maintenance under updates and adding an optimizer rule to exploit PatchIndex information in query optimization. In the following we describe the respective adaptions for integrating the patched multi-key partitioning constraint.

5.1 PatchIndex Creation

After running one of the graph partitioning algorithms described in Section 4 we create PatchIndexes on the partition key columns in order to maintain the exceptions to the partitioning. We follow the approach of having sets of patches per column, so we create separate index instances on them. The sets of patches maintain the tuple identifiers of all exceptions to the partitioning, i.e. all tuples that have a partition key value that is present in more than one partition or have a negative partition_id indicating a conflict when mapping the graph back to the table. We discover the partition key values violating the partition constraint by (1) running an aggregation query counting the distinct partition_ids of column values and selecting the ones having more than one distinct partition_id and (2) running a filter query to find all values with negative partition_ids. The union of (1) and (2) is joined back with the table on the respective key column to find all tuple identifiers to be inserted into the PatchIndex. In the upper case of Figure 7 all tuples with $B = 2$ are exceptions, so tuple identifiers of tuples $(1, 2)$ (assigned to the first partition) and $(2, 2)$ (assigned to the second partition) are inserted into the PatchIndex holding the set of patches P_B. Partitioning is transparent for the PatchIndex, so an index instance is created per partition.

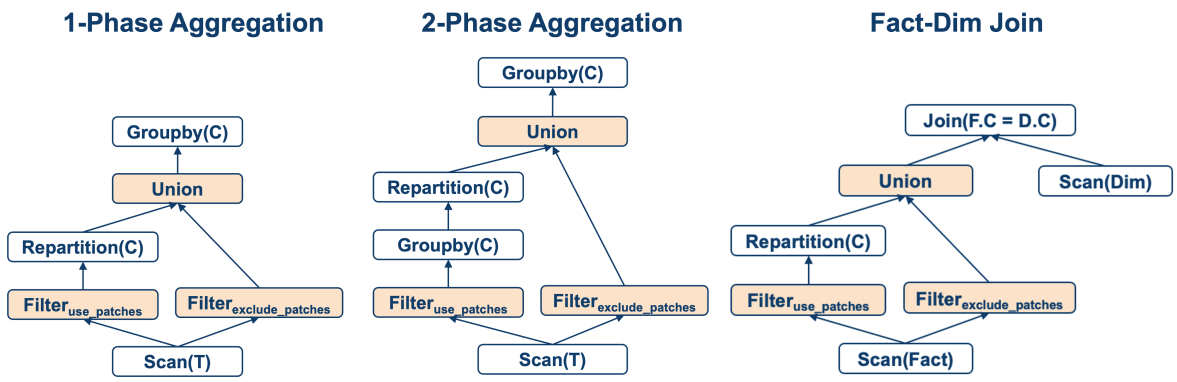

Figure 8: Patched query plans after applying the PatchIndex optimization rule (added operators highlighted)

5.2 Analytical Queries

Materializing approximate constraints with PatchIndexes follows the idea to split query execution using the PatchIndex scan between tuples that match a certain constraint and exceptions to the constraints[19]. The patched multi-key partitioning constraint defined in Definition 4.1 allows us to avoid expensive repartition/shuffle operations as we know that tuples not included in P_X for a given column X match the Partition locality criteria. Consequently, these tuples do not need to be reshuffled.

Partition locality plays an important role for aggregations and joins. For executing a partitioned aggregation, two basic approaches exist if the grouping column does not match the partition key of the table. First, data can be reshuffled on the grouping key and the subsequent aggregation can be executed partition-locally. This approach introduces large network overhead for repartitioning. Second, one could perform a pre-aggregation on the existing partitions, then reshuffle data and perform a post-aggregation afterwards, which is necessary as not all elements of a group are initially placed in the same partition. Here one needs to carefully consider the aggregation function, that is required to be decomposable, e.g. an average can't be executed in a stepwise fashion without breaking it into a sum and a count. The 2-phase aggregation is favorable when data contains few distinct values according to statistics, so the additional aggregation effort is amortized by the less effort to shuffle data. Both variants can be enhanced by the patched partitioning constraint. As shown in Figure 8, we can skip the repartitioning (for the 1-phase aggregation) or the pre-aggregation and repartitioning (for the 2-phase aggregation) for all tuples that are not included in the set of patches when aggregating on a patched partition key. This is possible because either all tuples or no tuples sharing the same partition key value are handled as exceptions. Consequently we reduce the number of tuples that need to be either reshuffled or pre-aggregated and reshuffled. Splitting the datastream however also introduces overhead. As the 2-phase aggregation is typically chosen when the grouping column only has a small amount of unique values, we expect that the benefit of the PatchIndex optimization might be too small to amortize the overhead of added operators. As the number of exceptions are known during query optimization, the costs of both plans with and without the optimization can be estimated and the optimizer can decide between them. Except adding linear costs for the filter operators with modes *use_patches* and *exclude_patches*, no changes are needed in the cost model of an arbitrary optimizer.

Join operators can also be performed in a partitioned way if tables are partitioned on the join key, otherwise an expensive repartitioning is needed again. In typical data warehouse applications, many joins are performed between fact tables and dimension tables. While the latter ones are typically partitioned on their primary keys, choosing the partition key of fact tables consisting of many foreign keys is typically difficult. When we included the foreign key in the patched multi-key partitioning and have a PatchIndex on the join column, we can again avoid the repartitioning for all tuples not included in the set of patches. This case however requires the second table to be co-partitioned, which is automatically ensured by using the same partitioning function in e.g. hash-partitioning, but is not guaranteed in our graph partitioning approach. We therefore derive the partitioning of the primary key side of the join from the fact table when the foreign key was included in the multi-key partitioning computation. Consequently, dimension key values that do not belong to the respective set of patches in the fact table derive the same partition_id, while exceptions are assigned using common hash partition to be co-partitioned with the fact table exceptions after repartitioning. Deriving the partitioning is a problem for shared dimensions, as only one fact table can be decisive for the dimension table partitioning. However, it is conceptually better to repartition a smaller dimension table than a large fact table.

In contrast to hash partitioning where data is not required to be reshuffled only if the partition key matches the grouping/join key, the described optimizations work with every key that was part of the patched multi-key partitioning. So instead of a fast query when hitting the right key and slow queries for all other keys in case of hash partitioning, we aim at reaching robust performance for aggregate/join queries on all keys of the multi-key partition key. The performance will depend on the amount of tuples that are categorized as exceptions and assigned to the sets of patches. Additionally, partition responsibilities of nodes do not change when querying different partition keys, so patched multi-key partitioning also works for shared-nothing architectures. Node-local buffers are not invalid for different queries, which is the major drawback of hierarchical partitioning approach described in Section 3.

The queries shown in Figure 8 are also allowed to have additional operators below the groupby/join operator, e.g. additional filters or projections. These additional operators would be replicated to both datastreams including/excluding patches, so they are applied to both datastreams separately. As a current limitation, these operators are not allowed to be aggregations or joins,

so the PatchIndex optimization rule is currently only applied on the lowest aggregation/join in the query tree.

5.3 Update Queries

Updatability is an important factor to make a concept applicable in practice. Therefore, we need to maintain the patched partitioning constraint also under update operations, i.e. inserts, modifies and deletes. The PatchIndex data structure itself is designed in a generic way. The underlying sharded bitmap structure supports update operations and we simply need to implement an interface to define how the set of patches is maintained under updates. Inserts are handled similar to the initial loading, so tuples get a partition_id assigned by the partition assignment algorithms described in Section 4. However, it might happen that an inserted tuple connects two connectivity components of the graph representation, meaning that partition key column values become exceptions. This can similarly happen for modifies on partition key values. We discover these new exceptions using a join of the inserted/modified tuples with the table itself, performing the same distinct aggregation than during initial partitioning to find values with more than one unique partition_id assigned. We keep the join small by only scanning the inserted/modified tuples on the one side and performing data pruning on the full table based on the observed join keys and small materialized aggregates [22] on the table, to avoid a full table scan. Delete operations do not connect graph components but might split components, which does not harm the partitioning constraint. In general update operations might degenerate the graph and can lead to loosing the optimality of the graph partitioning. By monitoring exception rates this might lead to a recomputation of the partitioning after some time.

6 EVALUATION

With our experiments we show that patched multi-key partitionings can be found in real-world datasets while comparing the presented partitioning approaches and highlight properties of data and algorithms that impact partition quality. Additionally we show opportunities for query performance and outline challenges for future work. We integrated the support for patched multi-key partitionings using PatchIndexes into the Actian VectorH 6.2 system, which is based on the x100 query engine[7]. The system runs on a four node cluster, each node consisting of a Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz, 256 GB RAM and a 10 GBit/s network interface. We use the PublicBI benchmark dataset [31] representing real user Tableau workbooks and picked the CommonGovernment workbook as an example. The workbook consists of 110 GB raw data and 43 queries including 13 different group-by columns. For performance evaluation we focus on these analytical queries, as the update mechanism is similar to and extensively evaluated in [20].

We measure execution times for partitioning using the following workflows. For graph partitioning we require the presence of the full dataset. Therefore we start with a pre-loaded table, perform the table to graph mapping, the graph partitioning and the graph to table mapping, before repartitioning the table on the determined partition_id. For the iterative partitioning approach we do not require the full table. We read a fixed flat file line by line and decide the partition assignment per tuple, so the order-sensitivity of the iterative approach does not have an impact on different runs in the experiments. We write the data with the

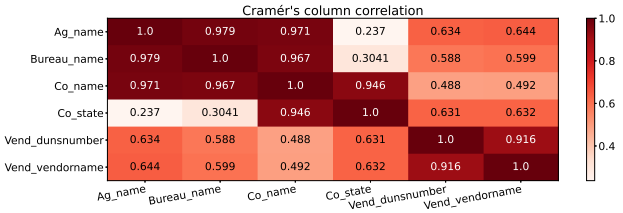

Figure 9: Column correlations in CommonGovernment

added partition_id back to a flat file and load the resulting table in a single bulk load.

Experiment 1: Partition quality. In the first experiment we show the existence of patched multi-key partitionings. We ran the partitioning on all pairs and triples based on the 13 grouping columns. Table 1 shows a subset of the candidates, chosen in order to present properties of our approach, with their respective statistics, i.e. partitioning runtime, imbalance factor and exception rates $e_{Col} = (|P_{Col}|/|R|) \cdot 100\%$ for both partition algorithms. All results show the median of three runs, which is particularly of interest for the graph partitioning algorithm starting at a random initial partitioning. We only use the graph partitioning with the $G_{t\_as\_e}$ mapping, as we observed that the selfjoin in $G_{t\_as\_v}$ leads to a very huge table when joining on columns with only a few unique values and out-of-memory situations. Table 1 shows different cases: The "ABC" columns *Ag_name*, *Bureau_name* and *Co_name* only have few hundred distinct values. Consequently, the resulting graph consists of few vertices with dense connectivity. In contrast, the "vendor" columns *Vend_dunsnumber* and *Vend_vendorname* have over 100K distinct values, leading to a graph of many vertices with sparse connectivity. In all cases, the number of edges is equal to the number of tuples by the definition of $G_{t\_as\_e}$.

We want to find a good partitioning indicated by both an imbalance factor near 1 and low exception rates for all columns. From the results we can outline properties that impact partition quality:

Column correlation: If columns correlate, they tend to have value combinations that are more likely than others. This leads to graphs with parts that are more densely connected than others and results in less exceptions when cutting edges from the graph. If columns are not correlated, the resulting graph has a more random distribution of edges, leading to a high number of exceptions. Figure 9 shows correlations of the nominal "ABC" and "vendor" columns based on the symmetric Cramér's V[9]. We can observe that correlating columns can lead to good partitionings in Table 1. *Co_state* does not correlate with *Ag_name* or *Bureau_name* columns and consequently leads to a bad partitioning when included.

Number of distinct values: As the number of edges is fixed by the number of tuples, the number distinct partition key values determine the number of vertices and consequently the degree of connectivity of the graph. We can can find a more balanced partitioning with less exceptions if there are many distinct partition key values in the data, which is the case for the "vendor" columns compared to the "ABC" columns.

Number of partitions: Similar to the number of distinct values, the number of desired partitions has an impact on partition quality. While we can find a reasonable good partitioning for the "ABC" columns with 4 partitions, running the algorithms for

Table 1: Runtime(in s), imbalance factor and exception rates(in %) of CommonGovernment table for different partitionings

Partition keys				1M Tuples						5M Tuples					
				Graph			Iterative			Graph			Iterative		
Column1	Column2	Column3	Parts	Time	Imbal.	Exc.	Time	Imbal.	Exc.	Time	Imbal.	Exc.	Time	Imbal.	Exc.
Ag_name	Bureau_name	-	4	46	4.9	(2,0)	39	2.3	(3,0)	100	2.2	(1,0)	175	2.5	(3,0)
Ag_name	Co_name	-	4	43	3.2	(10,1)	39	2.1	(10,1)	102	3.1	(43,1)	174	2.9	(23,1)
Bureau_name	Co_name	-	4	43	3.4	(1,0)	39	2.9	(3,0)	94	3.8	(21,1)	174	3.5	(3,0)
Vend_dunsnumber	Vend_vendorname	-	4	241	2.9	(0,1)	39	1.1	(16,23)	1050	3.1	(1,1)	179	1.01	(17,24)
Ag_name	Bureau_name	Co_name	4	49	2.6	(63,27,13)	40	2.3	(8,1,1)	121	4.2	(76,51,40)	178	2.5	(7,1,1)
Ag_name	Bureau_name	Co_state	4	47	2.4	(98,93,99)	40	1.2	(90,49,96)	110	2.7	(99,97,99)	176	1.4	(90,43,96)
Ag_name	Bureau_name	-	96	102	10K	(37,0)	60	15K	(2,0)	215	2.6K	(14,0)	198	22K	(3,0)
Ag_name	Co_name	-	96	109	13K	(82,1)	66	477	(12,1)	214	388K	(85,1)	199	680	(11,1)
Bureau_name	Co_name	-	96	118	13K	(33,1)	62	450	(3,0)	211	601K	(65,1)	198	557	(3,1)
Vend_dunsnumber	Vend_vendorname	-	96	207	21	(1,1)	63	6.1	(21,34)	560	28	(1,3)	206	8.5	(23,35)
Ag_name	Bureau_name	Co_name	96	117	111	(98,95,82)	60	128K	(8,1,1)	214	1.2	(100,100,100)	229	11K	(20,2,1)
Ag_name	Bureau_name	Co_state	96	114	905	(99,95,99)	63	5.5	(92,81,96)	215	1.3	(100,100,100)	223	5.6	(92,82,97)

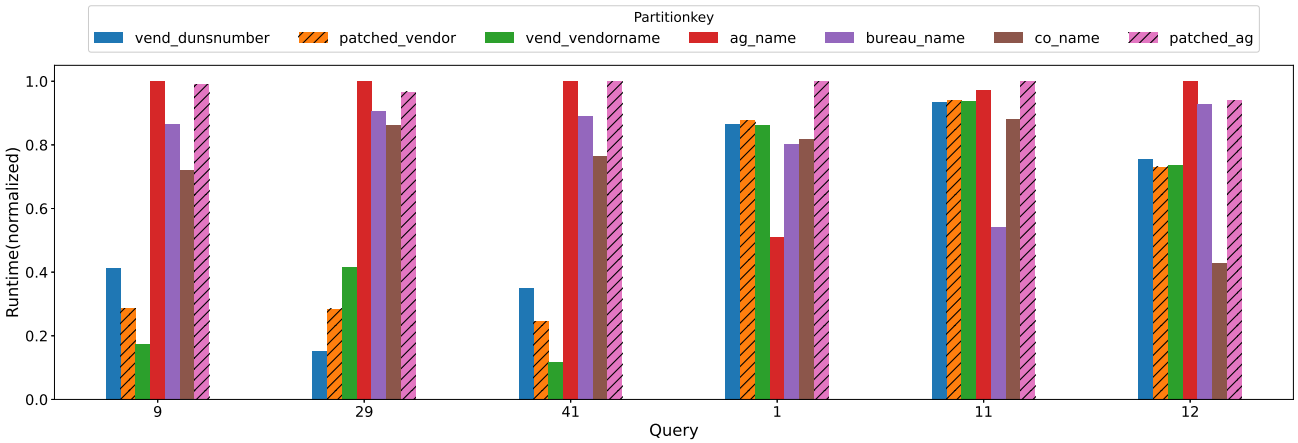

Figure 10: Query runtimes on different table partitionings

96 partitions leads to very bad partitionings indicated by either a high imbalance factor or high exception rates. Imbalance factor and exception rates are a trade-off here, as shown for, e.g., (*Ag_name*, *Bureau_name*) at 96 partitions and 5M tuples. The graph partitioning leads to higher exception rates but lower imbalance factor compared to the iterative approach. However, all partitionings with such imbalance factors are not usable. As the "vendor" columns contain more unique values and consequently more vertices in the graph, they are suitable to find a partitioning with more partitions. As a side note, the graph partitioning algorithm runs slower on the "vendor" columns for 4 partitions than for 96 partitions, as communication overhead for 4 partitions dominate the benefit of parallelism here.

Apriori property: From the results we can also suggest that the choice of partition key columns follow an apriori property. A partition key of k columns can only lead to a good partitioning if all $k-1$ subkeys provide a good partitioning. This is shown by the "ABC" columns in the iterative approach. While we get a good partitioning on (*Ag_name*, *Bureau_name*, *Co_name*), exchanging *Co_name* with *Co_state* results in a bad partitioning, as none of the subkeys containing *Co_state* is a good partitioning. Graph partitioning on three columns leads to high exception rates for the "ABC" columns due to the conflict resolution when mapping $G_{t\_as\_e}$ back to the table. Due to the apriori property the example given in Table 1 is limited to three partition key columns, as there are no more columns that correlate with the "ABC" columns in the dataset. Please note that this is a property of the dataset, not of our approach.

From a runtime perspective, the graph partitioning runtime is mainly dependent on the complexity of the graph, i.e. the number of vertices, as it propagates partition assignments over paths in the graph. Consequently, runtime on the "ABC" columns is significantly lower than on the "vendor" columns. The runtime thereby splits into several parts: The mapping from table to graph takes around 5-12s depending on the table size, running DistributedNE takes between 2-950s depending on the graph size, mapping the graph back to the table takes 4-9s and repartitioning the table takes 40-70s, so for small graph sizes repartitioning is the dominating part. On the other hand, iterative partitioning runtime is mainly impacted by the number of tuples as it iteratively processes the tuples. It is therefore constant for the same table size independent of the chosen partition keys. Runtime splits into iterating over the flat file and loading the resulting flat file into the database, which took around 20s for 1M tuples or 100s for 5M tuples. The iterative approach is favorable over the graph partitioning in the currently presented version, both in terms of runtime and partitioning quality.

This experiment also reveals a weakness of the current approach. Starting from a good partitionining (*Ag_name*, *Bureau_name*), adding *Co_state* destroys the whole partitioning instead of just excluding the values from the uncorrelated *Co_state* column. This is caused by the fact that the underlying partitioning algorithm is not aware that the graph is multipartite, i.e. it has subsets of vertices (one subset per column) and all edges are between subsets, but never within a subset. For our application, it would be better to simply cut a whole subset instead of destroying the whole partitioning. In the example, it would be better to have 100% exceptions in *Co_state* while keeping the good partitioning in (*Ag_name*, *Bureau_name*). With this, queries that require a partitioning on the column

Table 2: TPC-DS partition key column combinations and exception rates

Table	Partition Key Columns	Exception Rates
Item	(i_item_sk, i_item_desc, i_formulation, i_product_name)	(0%, 0.2%, 0%, 0.01%)
Item	(i_item_sk, i_item_desc, i_formulation, i_product_name, i_manufact_id, i_manufact)	(4.7%, 5.1%, 5.1%, 0%, 83%, 67%)
Store_returns	(sr_customer_sk, sr_cdemo_sk, sr_ticket_number)	(3.8%, 0.07%, 3.7%)
Catalog_returns	(cr_refunded_customer_sk, cr_refunded_cdemo_sk, cr_returning_cdemo_sk, cr_order_number)	(3.7%, 1.7%, 2.1%, 0.6%)
Catalog_returns	(cr_refunded_addr_sk, cr_returning_cdemo_sk, cr_order_number)	(5.2%, 2.4%, 1.1%)
Web_returns	(wr_item_sk, wr_refunded_cdemo_sk, wr_returning_cdemo_sk)	(5.1%, 0.8%, 0.7%)
Web_returns	(wr_refunded_customer_sk, wr_refunded_cdemo_sk, wr_returning_customer_sk, wr_returning_cdemo_sk)	(0.4%, 0.4%, 0.5%, 0.4%)
Web_returns	(wr_refunded_cdemo_sk, wr_refunded_hdemo_sk, wr_returning_cdemo_sk, wr_returning_hdemo_sk)	(1.1%, 6.2%, 1.1%, 6.2%)
Web_returns	(wr_refunded_cdemo_sk, wr_refunded_addr_sk, wr_returning_cdemo_sk, wr_returning_addr_sk)	(0.7%, 1.0%, 0.7%, 1.0%)
Web_sales	(ws_bill_customer_sk, ws_bill_cdemo_sk, ws_bill_addr_sk, ws_ship_customer_sk, ws_ship_cdemo_sk, , ws_order_number)	(13.7%, 0.7%, 22.8%, 13.2%, 0.8%, 0%)
Web_sales	(ws_bill_customer_sk, ws_bill_cdemo_sk, ws_ship_customer_sk, ws_ship_cdemo_sk, ws_order_number)	(7.3%, 0.4%, 7.3%, 0.4%, 0%)
Catalog_sales	(cs_bill_customer_sk, cs_bill_cdemo_sk, cs_ship_customer_sk, cs_ship_cdemo_sk, cs_order_number)	(8.1%, 1.7%, 8.1%, 1.6%, 0.1%)
Catalog_sales	(cs_bill_cdemo_sk, cs_bill_addr_sk, cs_ship_cdemo_sk, cs_ship_addr_sk, cs_order_number)	(1.8%, 17.3%, 17.3%, 17.3%, 0.1%)
Store_sales	(ss_customer_sk, ss_cdemo_sk, ss_ticket_number)	(10.2%, 3.8%, 0.5%)
Store_sales	(ss_cdemo_sk, ss_addr_sk, ss_ticket_number)	(3.4%, 18.8%, 1.2%)

cutted by the algorithm would run similar than before using repartitioning, but queries on the remaining columns would be accelerated.

Experiment 2: Query performance. Out of all queries of the CommonGovernment workbook we chose two groups. Queries 9, 29 and 41 contain aggregations on *Vend_dunsnumber* and *Vend_vendorname*, while queries 1, 11 and 12 contain aggregations on *Ag_name*, *Bureau_name* and *Co_name*. We loaded the full dataset with hash partitionings on each of these columns as baselines and compared them against patched multi-key partitionings using the iterative partitioner on *(Vend_dunsnumber, Vend_vendorname)* named as *patched_vendor* and *(ag_name, bureau_name, co_name)* named as *patched_ag*, which were the column combinations with useful partitionings from the first experiment. All tables have 96 partitions. The results are shown in Figure 10. We can first observe that the baselines show different runtimes, although they are expected to perform similarly. Partitionings on the vendor columns are faster than partitionings on *Ag_name*, *Bureau_name* or *Co_name*, caused by the small amount of unique values of the latter columns leading to imbalanced partitionings.

Overall the partitioning that matches the grouping column of the respective query shows the fastest runtime as expected. For the first set of queries (9, 29, 41), the *patched_vendor* partitioning shows performance slightly worse than the best partitioning, but better than the respective other vendor partition key. This is exactly the behaviour we want to achieve. We achieve a consistent performance with a patched multi-key partitioning for this set of queries.

For the second set of queries (1, 11, 12) we can observe a different behaviour. Here, the patched multi-key partitioning is not able to achieve reasonable good performance cuased by the small number of unique values of the partitioning columns. The aggregations are performed as a two-phase aggregation and the overhead of adding a PatchIndex scan is too large to amortize it's benefits. Additionally, partitions are even more imbalanced than in the respective baselines and the exception rates are very high as shown in the first experiment. Consequently, using a patched multi-key partitioning on partition keys with small number of distinct values (leading to bad imbalance factors and high exception rates) does not show the robust performance we are aiming at and should not be used.

Besides that, robustness also means reasonable performance in cases where partition keys do not match grouping keys. We can observe that the *patched_vendor* partitioning behaves similar than the other vendor partitionings in the second set of queries (1, 11, 12), while the *patched_ag* partitioning performs similarly than the partitionings on the "ABC" columns on the first set of queries (9, 29, 41). In conclusion we can state that patched multi-key partitioning shows robust performance for columns with a quite large amount of unique values. It avoids bad query performance caused by full table repartitionings when matching the grouping keys while performing similar to the baselines when not matching the grouping key. Please note that the case of replication is also covered in the experiment, i.e. one could achieve best performance in all queries when holding replicas for all partition keys leading to 5x storage and update cost.

Experiment 3: TPC-DS. In order to show the robustness of our approach over different datasets, we investigated the well-known TPC-DS[26] benchmark dataset. Although being synthetic and therefore rarely contatining "natural" clusters we found different useful partition key combinations in the dataset on scale factor 1. We used the column correlation requirement and the apriori property to find the combinations: In the fact tables we calculated the column correlations of all interesting columns (i.e. foreign keys or nominal values) and searched for combinations of columns that are all pairwise correlated with a correlation of over 0.9 and fulfill the apriori property. The resulting column combinations are candidates and were partitioned using the iterative partitioning algorithm into 4 partitions. Table 2 shows several partitionings with their respective exception rates, all showing a partition balance factor below 1.1.

7 CONCLUSION

We introduced the concept of patched multi-key partitioning as a solution for the multi-key partitioning problem of relational tables. In value-based partitioning like the commonly used hash partitioning choosing multiple partition keys is not possible without replicating data. We described that hierarchical partitioning is no reasonable solution to the problem and therefore defined the concept of patched multi-key partitioning, which is based on handling exceptions to the partitioning constraint using e.g. PatchIndexes. By mapping relational tables to graphs we can use existing graph partitioning algorithms to find a patched partitioning. We described how patched multi-key partitioning can be exploited in query execution and showed the opportunities of the approach to achieve robust query performance for workloads requiring different partitionings in our evaluation. We think patched multi-key partitioning is a promising concept for distributed query processing. In the future, we aim at designing a graph algorithm that solves the challenges stated in the evaluation, especially being aware that the graph is multipartite. Additionally, the approach should be extended to whole database schemas, i.e. propagating the partitioning information over foreign-key related tables to enable co-partitioned joins.

REFERENCES

[1] Daniel Abadi, Peter Boncz, and Stavros Harizopoulos. 2013. *The Design and Implementation of Modern Column-Oriented Database Systems.* Now Publishers Inc., Hanover, MA, USA.

[2] Hamilton Wilfried Yves Adoni, Tarik Nahhal, Moez Krichen, Brahim Aghezzaf, and Abdeltif Elbyed. 2020. A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems. *Distributed and Parallel Databases* 38, 2 (June 2020), 495–530. https://doi.org/10.1007/s10619-019-07276-9

[3] V. E. Alekseev, R. Boliac, D. V. Korobitsyn, and V. V. Lozin. 2007. NP-hard graph problems and boundary classes of graphs. *Theoretical Computer Science* 389, 1 (2007), 219–236. https://doi.org/10.1016/j.tcs.2007.09.013

[4] Brian Babcock and Surajit Chaudhuri. 2005. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/1066157.1066172

[5] Stephan Baumann, Peter A. Boncz, and Kai-Uwe Sattler. 2016. Bitwise dimensional co-clustering for analytical workloads. *The VLDB Journal* 25, 3 (June 2016), 291–316. https://doi.org/10.1007/s00778-015-0417-y

[6] Bishwaranjan Bhattacharjee, Sriram Padmanabhan, Timothy Malkemus, Tony Lai, Leslie Cranston, and Matthew Huras. 2003. Efficient Query Processing for Multi-Dimensionally Clustered Tables in DB2. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29 (VLDB '03)*. VLDB Endowment, 963–974. event-place: Berlin, Germany.

[7] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[8] Aydin Buluc, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2015. Recent Advances in Graph Partitioning. *arXiv:1311.3144 [cs, math]* (Feb. 2015). http://arxiv.org/abs/1311.3144

[9] Harald Cramér. 1946. *Mathematical Methods of Statistics.* Princeton University Press. https://books.google.de/books?id=_db1jwEACAAJ

[10] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 48–57. https://doi.org/10.14778/1920841.1920853 Publisher: VLDB Endowment.

[11] Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. 2020. Incrementalization of Graph Partitioning Algorithms. *Proc. VLDB Endow.* 13, 8 (2020), 1261–1274. https://doi.org/10.14778/3389133.3389142

[12] Carlos Garcia-Alvarado, Venkatesh Raghavan, Sivaramakrishnan Narayanan, and Florian Waas. 2012. Automatic Data Placement in MPP Databases. In *2012 IEEE 28th International Conference on Data Engineering Workshops*. 322–327. https://doi.org/10.1109/ICDEW.2012.45

[13] Goetz Graefe, Wey Guy, Harumi A. Kuno, and G. N. Paulley. 2012. Robust Query Processing (Dagstuhl Seminar 12321). *Dagstuhl Reports* 2 (2012), 1–15.

[14] Alessio Guerrieri. 2015. Distributed computing for large-scale graphs. *PhD Thesis* (2015), 141.

[15] Masatoshi Hanai, Toyotaro Suzumura, Wen Jun Tan, Elvis Liu, Georgios Theodoropoulos, and Wentong Cai. 2019. Distributed Edge Partitioning for Trillion-Edge Graphs. *Proc. VLDB Endow.* 12, 13 (Sept. 2019), 2379–2392. https://doi.org/10.14778/3358701.3358706

[16] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, Portland OR USA, 143–157. https://doi.org/10.1145/3318464.3389704

[17] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive Query Processing on RAW Data. *Proc. VLDB Endow.* 7, 12 (aug 2014), 1119–1130. https://doi.org/10.14778/2732977.2732986

[18] George Karypis and Vipin Kumar. 1999. Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing 20(1), 359-392. *Siam Journal on Scientific Computing* 20 (1999). https://doi.org/10.1137/S1064827595287997

[19] Steffen Kläbe, Kai-Uwe Sattler, and Stephan Baumann. 2020. PatchIndex - Exploiting Approximate Constraints in Self-managing Databases. In *36th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 139–146. https://doi.org/10.1109/ICDEW49219.2020.00014

[20] Steffen Kläbe, Kai-Uwe Sattler, and Stephan Baumann. 2021. Updatable Materialization of Approximate Constraints. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1991–1996. https://doi.org/10.1109/ICDE51399.2021.00189

[21] Dong Liming, Liu Weidong, and Shao Jie. 2017. Coexistence of Multiple Partition Plan Based Physical Database Design. In *Proceedings of the 5th International Conference on Communications and Broadband Networking (ICCBN '17)*. Association for Computing Machinery, New York, NY, USA, 41–46. https://doi.org/10.1145/3057109.3057111 event-place: Bali, Indonesia.

[22] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*.

[23] Rimma Nehme and Nicolas Bruno. 2011. Automated Partitioning Design in Parallel Database Systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 1137–1148. https://doi.org/10.1145/1989323.1989444 event-place: Athens, Greece.

[24] Panos Parchas, Yonatan Naamad, Peter Van Bouwel, Christos Faloutsos, and Michalis Petropoulos. 2020. Fast and Effective Distribution-Key Recommendation for Amazon Redshift. *Proc. VLDB Endow.* 13, 11 (2020), 2411–2423. http://www.vldb.org/pvldb/vol13/p2411-parchas.pdf

[25] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2213836.2213844 event-place: Scottsdale, Arizona, USA.

[26] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why You Should Run TPC-DS: A Workload Analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 1138–1149.

[27] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, and Seif Haridi. 2014. Distributed Vertex-Cut Partitioning. In *Distributed Applications and Interoperable Systems*, Kostas Magoutis and Peter Pietzuch (Eds.). Vol. 8460. Springer Berlin Heidelberg, Berlin, Heidelberg, 186–200. https://doi.org/10.1007/978-3-662-43352-2_15

[28] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. 2015. A Distributed Algorithm for Large-Scale Graph Partitioning. *ACM Transactions on Autonomous and Adaptive Systems* 10 (2015), 1–24. https://doi.org/10.1145/2714568

[29] Thomas Stöhr, Holger Märtens, and Erhard Rahm. 2000. Multi-Dimensional Database Allocation for Parallel Data Warehouses. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 273–284.

[30] Ramakrishna Varadarajan, Vivek Bharathan, Ariel Cary, Jaimin Dave, and Sreenath Bodagala. 2014. DBDesigner: A customizable physical design tool for Vertica Analytic Database. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 1084–1095. https://doi.org/10.1109/ICDE.2014.6816725

[31] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, Alexander Böhm and Tilmann Rabl (Eds.). ACM, 1:1–1:6. https://doi.org/10.1145/3209950.3209952

[32] Marianne Winslett and Vanessa Braganholo. 2020. Goetz Graefe Speaks Out on (Not Only) Query Optimization. *SIGMOD Rec.* 49, 3 (dec 2020), 30–36. https://doi.org/10.1145/3444831.3444837

[33] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. 2015. Robust Query Optimization Methods With Respect to Estimation Errors: A Survey. *SIGMOD Rec.* 44, 3 (dec 2015), 25–36. https://doi.org/10.1145/2854006.2854012

[34] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. 2015. Locality-Aware Partitioning in Parallel Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/2723372.2723718 event-place: Melbourne, Victoria, Australia.