CONTENTS

Robust and Memory-Efficient Database Fragment Allocation for Large and Uncertain Database Workloads

Rainer Schlosser*
Hasso Plattner Institute, Potsdam, Germany
rainer.schlosser@hpi.de

Stefan Halfpap*
Hasso Plattner Institute, Potsdam, Germany
stefan.halfpap@hpi.de

ABSTRACT

Database replication and query load-balancing are mechanisms to scale query throughput. The analysis of workloads allows load-balancing queries to replica nodes according to their accessed data. As a result, replica nodes must only store and synchronize subsets of the data. However, balancing the load of large-scale workloads evenly while minimizing the memory footprint is complex and challenging. State-of-the-art allocation approaches are either time consuming or the resulting allocations are not memory-efficient. Further, partial replication approaches usually optimize only against a single fixed workload. If the actual workload deviates from this expected one, load-balancing can be highly skewed, resulting in severe performance degradation.

This paper proposes a novel approach to compute memory-efficient fragment allocations that enable balancing multiple potential future workloads. Applied on the TPS-DS benchmark and a large-size enterprise workload, we show that, compared to state-of-the-art allocations, our solutions are (i) more flexible, (ii) require up to 50% less data, (iii) have competitive runtimes, and (iv) are more robust against uncertain out-of-sample workloads.

1 INTRODUCTION

Increasing demand for database processing capabilities can be managed by scale-out approaches, using additional servers. Analyses of enterprise workloads have shown that both OLTP and OLAP are read-dominant [9]. Database replication is an approach to scale-out read-only queries, which can be executed on snapshots of a primary server without violating consistency [4].

Given that most queries require only a limited set of tuples and attributes, partial replication is an efficient approach: Instead of duplicating all data to all replica nodes, partial replicas store only a subset of the data while being able to process a large workload share. Thereby, splitting the overall workload evenly among the replica nodes is essential to scale the query throughput linearly with the number of nodes. Partial replication consists of two steps, which are typically separated from each other to better deal with the problem complexity [11]. First, the data set is partitioned horizontally and/or vertically into disjoint data partitions/fragments. Second, the individual fragments are allocated to one or multiple nodes.

This paper addresses the second step, i.e., a fragment allocation problem (given a fixed data partitioning). Calculating efficient fragment allocations that minimize the replicas' memory consumption while evenly balancing the query load is challenging, because the data allocation and the workload distribution are mutually dependent. As the calculation of optimal allocations is an NP-hard problem, heuristic approaches have to be used for large problem sizes. Rabl and Jacobsen propose a greedy allocation approach with short computation times [12]. We have previously proposed a decomposition approach [5] based on linear programming (LP), which calculates allocations with up to 23% lower memory consumption for the TPC-H benchmark.

Allocations for larger workloads are harder to solve, but typically offer greater potential for sophisticated approaches compared to simple heuristics. However, when using the LP-based decomposition approach for larger problems, computation times increase, and problems may finally become practically intractable, e.g., for an application in dynamic settings, in which model inputs change and quick recalculations are required. Considering multiple potential workloads to increase an allocation's robustness increases the problem complexity even further. However, such robustness is necessary in practice, when workloads fluctuate, and query costs or frequencies cannot be predicted precisely.

The goal of this paper is to overcome the limitations of existing allocation approaches. Applied to TPC-DS and a real-world accounting workload, we show that the greedy rule-based heuristic [12] is not memory-efficient, while the solver-based solution [5] provides low robustness against diversified workloads and has unacceptable runtimes to be used in practice. To fill this gap, we propose a heuristic LP-based clustering approach to *flexibly combine robustness, memory-efficiency, and a short calculation time* for large-scale problems. Our *contributions* are the following:

First, we derive robust and memory-efficient fragment allocations, which enable an even load balancing against multiple potential future workloads. Second, exploiting the skewness of workloads, we use partial clustering techniques to compute solutions for large-scale workloads quickly. Third, for the TPC-DS and a large real-world workload, we show that, with our techniques, the trade-off between memory-efficiency, robustness, and a short calculation time can be smoothly balanced in a targeted way. Fourth, we verify the robustness of our allocations by confronting them with unseen out-of-sample workloads.

2 FRAGMENT ALLOCATION PROBLEM AND LIMITS OF EXISTING APPROACHES

2.1 Problem Description and Difficulty

The scale-out of workloads to partially replicated databases leads to a coupled data assignment and workload distribution problem. We consider a horizontally and/or vertically partitioned database consisting of N disjoint fragments.

The workload input can be described as follows. We consider the case where data (fragments) can be stored on K nodes to distribute a given workload. The size of a fragment i is a_i, $i = 1, ..., N$. Further, we consider a set of Q queries j, characterized by fragments accessed, i.e., $q_j \subseteq \{1, ..., N\}$, $j = 1, ..., Q$. We assume that the costs of queries j are independent of the executing node k, $k = 1, ..., K$, and determined by c_j, $j = 1, ..., Q$. Query costs are numerical and can, e.g., be modeled as average processing times. We assume that the queries j occur with a given frequency f_j, $j = 1, ..., Q$. The total workload costs C are $C := \sum_{j=1,...,Q} f_j \cdot c_j$.

*Both authors contributed equally to this research.

The allocation problem can be described as follows. The goal is to decide (i) on which node to put which fragments and (ii) which query is executed at which node to which extent (workload share). Our objective is to *minimize* the total amount of data at all nodes such that the workload can be evenly distributed between them. Further: (i) A query j can only be executed at node k if all relevant fragments are stored on node k. (ii) For each of the Q queries, the workload shares, assigned to the different nodes, have to sum up to one. (iii) At each of the K nodes, the workload share has to be $1/K$ such that the overall query throughput can be scaled. (iv) Further, as high calculation times may limit the applicability in practice, we want to compute optimized allocations quickly. (v) The allocation should work for multiple given workload scenarios and should be robust against new unseen ones.

2.2 Existing Fragment Allocation Approaches

2.2.1 Optimal Solution via Linear Programming. For a single given workload, the described fragment allocation problem can be formulated as a linear mixed integer problem, cf. [5, 12]. However, the complexity of the linear program quickly increases with the number of queries (Q), fragments (N), and nodes (K). For this reason, the optimal solution can only be derived as long as the size of the problem is sufficiently *small*.

2.2.2 Greedy Heuristic. Rabl and Jacobsen [12] start to assign queries with the largest workload share and accessing the most data. Queries are ordered by the product of the workload share and the total size of accessed fragments. A query is assigned to the node with the largest overlap of already allocated fragments and those accessed by the query. Nodes with no assigned queries are thereby treated as if they have a complete overlap. If a query's workload share exceeds the assigned node's load capacity, the node is filled up to its limit. The query with its remaining workload is merged back into the list of queries and assigned later.

2.2.3 LP-Based Decomposition. We proposed to iteratively *split* the workload into smaller workload packages (chunks) such that the data redundancy is minimized in each step [5]. In a split with B subnodes, each subnode b represents n_b final nodes, $b = 1, ..., B$, and takes the workload share $w_b := n_b/K$. The special case $B = K$ corresponds to the optimal solution, cf. Section 2.2.1. The workload splits are obtained using small-sized LP *subproblems* similar to the LP structure of optimal solutions. In the LP, the following variables are used: The binary variables $x_{i,k} \in \{0, 1\}$, $i = 1, ..., N$, $k = 1, ..., K$, indicate whether fragment i is allocated to node k (1) or not (0). The binary variables $y_{j,k} \in \{0, 1\}$, $j = 1, ..., Q$, $k = 1, ..., K$, indicate whether query j can run on node k (1) or not (0). The continuous variables $z_{j,k} \in [0, 1]$, $j = 1, ..., Q$, $k = 1, ..., K$, represent the workload share of query j executed at node k. The sum of shares has to sum up to one for all queries. Further, by W/V, the *replication factor* is denoted, where the total amount of data used

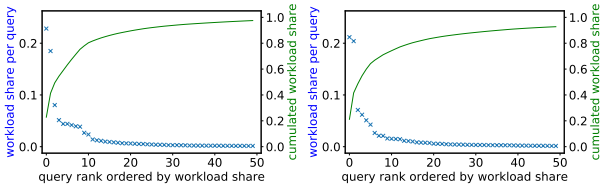
$$W := \sum_{i=1,..,N,k=1,...,K} x_{i,k} \cdot a_i \tag{1}$$

is normalized by the amount of overall accessed data

$$V := \sum_{i \in \bigcup_{j=1,...,Q: f_j > 0} \{q_j\}} a_i. \tag{2}$$

2.3 Large and Skewed Workloads

The results of [5] and [12] were shown for the TPC-H benchmark, which consists of $Q = 22$ queries and $N = 61$ fragments/columns. We use the more complex TPC-DS benchmark ($Q = 99$, $N = 425$) and a real-world enterprise workload ($Q = 4\,461$, $N = 344$).



(a) TPC-DS **(b) Accounting workload**

Figure 1: Distribution of top 50 query workload shares in decreasing order, cf. (a) Section 2.3.1 and (b) Section 2.3.2.

2.3.1 TPC-DS Workload. To obtain model inputs, we loaded the tables with scale factor 1 into a PostgreSQL 12.2 database system. We use vertical partitioning with each column as an individual fragment. We deployed single column indices on all primary key columns. Fragment sizes a_i, $i = 1, ..., 425$, are modeled by using the function `pg_column_size()` to calculate the pure value sizes, abstracting from the PostgreSQL page layout with meta-information and padding. In case the column is part of a primary key, the index size increases the associated fragment size. We use the command `pg_table_size(index_name)` to calculate index sizes. We derived query costs c_j as average execution time for query template j with varying parameters. For TPC-DS queries 1, 4, 6, 11, and 74, the set timeout of 120 s was exceeded. Thus, we omitted them in our experiments, resulting in $Q = 94$ queries.

2.3.2 Real-World Accounting Workload. We got access to metadata of an enterprise's central accounting table and a summary of a workload trace against this table in the form of query templates and statistics. The metadata enabled us to derive all required model inputs for calculating fragment allocations using vertical partitioning. The anonymized workload metadata and source code to reproduce the allocations are publicly available online [1]. The analyzed table stores accounting information and has N=344 columns. The summary of the workload trace consists of Q=4 461 SQL templates with aggregated execution properties of individual queries. Thereby, the most important properties for our research are query frequencies f_j (occurrences) and costs c_j, i.e., the average execution time per query (template).

2.3.3 Workload Skewness. Figure 1 shows the distribution of and cumulative query workload shares $f_j \cdot c_j$, $j = 1, ..., Q$. For both workloads, the distribution is *highly skewed*: The queries with the 50 highest workload shares account for more than 97% of the TPC-DS and more than 92% of the real-world workload.

2.4 Limitations of Existing Approaches

To study the suitability of existing approaches, we calculated allocations for TPC-DS and a real-world workload, cf. Section 2.3. For different numbers K, Table 1 summarizes memory consumption and runtime of existing allocations approaches. We used $f_j := 1$, for all $j = 1, ..., Q$. The LP-results of [5] were achieved using the Gurobi solver (version 9.0.0) (single-threaded). For the comparison with [12]'s approach, we implemented their algorithms in Python 3, and declare the runtime as an upper limit (<).

The upper part of both subtables shows the memory consumption (W/V) and required runtime for the optimal solution, cf. W^D without chunking, for up to $K = 6$ (TPC-DS) and $K = 5$ (accounting workload) nodes compared to the greedy heuristic, cf. W^G. We could not compute optimal allocations for larger numbers of nodes K within 8 hours. The optimal replication factors provide a useful reference to verify the quality of heuristic approaches. In this context, we observe that the greedy heuristic requires up to 100% more data than the optimal solution.

Table 1: Advantages and disadvantages of existing approaches: the greedy heuristic [12] (W^G), cf. Sec. 2.2.2, vs. the decomposition heuristic [5] (W^D), cf. Sec. 2.2.3, including optimal solutions, cf. Sec. 2.2.1, (*no decomposition).

(a) TPC-DS; $K = 2, ..., 12$, $N = 425$, $Q = 94$.

K	chunks	$\frac{W^D}{V}$	solve time$_{W^D}$	$\frac{W^G}{W^D}$	solve time$_{W^G}$
2	2	1.126*	1 s	+1%	<0.1 s
3	3	1.205*	9 s	+53%	<0.1 s
4	4	1.298*	43 s	+94%	<0.1 s
5	5	1.393*	407 s	+88%	<0.1 s
6	6	1.457*	1074 s	+100%	<0.1 s
4	2+2	1.310	2 s	+93%	<0.1 s
5	3+2	1.466	14 s	+79%	<0.1 s
6	3+3	1.519	14 s	+92%	<0.1 s
8	4+4	1.874	122 s	+65%	<0.1 s
10	5+5	2.076	517 s	+61%	<0.1 s
12	6+6	2.201	2 146 s	+60%	<0.1 s

(b) Real-world workload; $K = 2, ..., 12$, $N = 344$, $Q = 4\,461$, cf. source [1].

K	chunks	$\frac{W^D}{V}$	solve time$_{W^D}$	$\frac{W^G}{W^D}$	solve time$_{W^G}$
2	2	1.322*	179 s	+51%	<3 s
3	3	1.775*	6 236 s	+50%	<3 s
4	4	2.104*	9 356 s	+74%	<3 s
5	5	2.473*	13 738 s	+57%	<3 s
3	2+1	1.811	384 s	+47%	<3 s
4	2+2	2.126	225 s	+72%	<3 s
5	2+2+1	2.499	5 922 s	+55%	<3 s
6	3+3	2.855	778 s	+59%	<3 s
8	3+3+2	3.499	8 859 s	+87%	<3 s
10	4+3+3	4.462	49 233 s	+74%	<3 s
12	4+4+4	5.162	47 207 s	+82%	<3 s

The lower part of the subtables shows the results of the decomposition heuristic compared to the greedy heuristic. The decomposition and greedy approach make it possible to solve the problem for larger K heuristically. We observe that the decomposition approach (W^D) yields better replication factors than the greedy approach (W^G), which requires up to 93% and 87% more data for TPC-DS and the accounting workload, respectively. Further, the decomposition approach performs close to optimal compared to the optimal solution results (for $K \leq 6$, see Table 1a and for $K \leq 5$, see Table 1b). However, the decomposition approach requires high computation times if the problem becomes large, e.g., in the case of the accounting workload. In contrast, the greedy approach is fast and requires only seconds.

2.5 Uncertain Future Workloads

In general, future workloads are not entirely predictable. Thus, a further weakness of existing approaches is that they are only optimized for a single workload and can perform poorly if actual workloads differ. Hence, it is crucial to take potential workload scenarios into account to obtain a robust performance. Potential future workload scenarios can be determined, e.g., based on previously observed (seasonal) workloads or forecasts. In this context, fragment allocations should be such that the workload can be successfully balanced in any scenario. We assume that a workload scenario is characterized by a set of queries with given frequencies and costs within a certain time span.

In [12], Rabl and Jacobsen also describe an extension of their approach to cope with multiple workload scenarios. They propose to calculate a separate allocation for each scenario independently. Individual allocations are merged pairwise, mapping each node of the first allocation to a node of the second allocation. A merged allocation enables an even load balancing for both input allocations. The Hungarian method allows calculating an optimal mapping, which minimizes the memory consumption of the merged allocation (in polynomial time). However, because entire nodes are merged, optimization potential is given away.

Overall, we observe that existing approaches have different strengths and weaknesses (runtime vs. memory-efficiency). Further, from a practical perspective, solutions are required that can provide a *reasonable combination* of (i) memory-efficiency, (ii) robustness against different workloads, and (iii) short runtimes. Our goal is to design a heuristic approach that is of that kind.

3 ROBUST FRAGMENT ALLOCATION FOR MULTIPLE POTENTIAL WORKLOADS

3.1 LP-Based Robust Solution Approach

In the following, we consider S potential workload scenarios. Each scenario s, $s = 1, ..., S$, is characterized by query frequencies $f_{j,s}$ and associated workload costs $C_s := \sum_{j=1,...,Q} f_{j,s} \cdot c_j$, cp. Section 2.1. Note, in this framework, also uncertain query costs c_j can be expressed similarly by using potential scenario-based costs $c_{j,s}$ without increasing the model's complexity.

The core idea is finding a single allocation that enables an even load balancing for all S potential workloads scenarios. Further, by enabling an even load balance for specific diversified scenarios, such enriched allocations also allow improved load balancing for *unseen* scenarios, which may be similar to mixtures of input scenarios. Thereby, an allocation's robustness can be increased by choosing a larger number of diverse scenarios.

Our robust multi-scenario model is an extension of the LP-based decomposition approach, cf. Section 2.2.3, which considers one deterministic workload, cf. $S = 1$. Compared to [5], we use extended variables (cf. z), to model workload shares for each scenario s, $s = 1, ..., S$. Based on the decomposition concept of [5], we propose the following extended LP model to allocate data fragments and to distribute workload shares to multiple nodes in the presence of multiple potential workloads:

$$\begin{aligned} &\textit{minimize} \\ &x_{i,b}, y_{j,b} \in \{0,1\}, z_{j,b,s} \in [0,1], 0 \leq L \leq 1, (\text{given } \bar{x}_i, \bar{y}_j, \bar{z}_{j,s}) \\ &i = 1, ..., N,\ j = 1, ..., Q,\ b = 1, ..., B,\ s = 1, ..., S \end{aligned}$$

$$\frac{1}{V} \cdot \sum_{i=1,..,N, b=1,...,B: \bar{x}_i=1} x_{i,b} \cdot a_i + \alpha \cdot L \tag{3}$$

$$s.t. \quad y_{j,b} \cdot |q_j| \leq \sum_{i \in q_j} x_{i,b}, \qquad \begin{matrix} j = 1, ..., Q : \bar{y}_j = 1 \\ b = 1, ..., B \end{matrix} \tag{4}$$

$$z_{j,b,s} \leq y_{j,b}, \qquad \begin{matrix} j = 1, ..., Q : \bar{y}_j = 1 \\ b = 1, ..., B, \quad s = 1, ..., S \end{matrix} \tag{5}$$

$$\sum_{j=1,...,Q: \bar{y}_j=1} f_{j,s} \cdot c_j / C_s / w_b \cdot z_{j,b,s} \leq L, \qquad \begin{matrix} b = 1, ..., B \\ s = 1, ..., S \end{matrix} \tag{6}$$

$$\sum_{b=1,...,B} z_{j,b,s} = \bar{z}_{j,s}, \qquad \begin{matrix} j = 1, ..., Q : \bar{y}_j = 1 \\ s = 1, ..., S \end{matrix} \tag{7}$$

The overall idea is that the model allocates the fragments such that the workload can be distributed evenly for all workload scenarios s, $s = 1, ..., S$. Note, scenario probabilities are not used and thus do not have to be quantified in advance.

The objective (3) minimizes the replication factor W/V, cp. (1) - (2). Constraint (4) guarantees that a query j can only be executed on node b if all relevant fragments are available, see Section 2.1. The cardinality term $|q_j|$ expresses the number of fragments used in query j. Constraint (5) ensures that a query j can only have a positive workload share on node b in scenario s if it can be executed on node b. Constraint (6) guarantees that,

in all scenarios s, all nodes b do not exceed the workload limit L. Here the workload is normalized by the total workload cost C_s of scenario s and the workload share $w_b := n_b/K$, cf. Sec. 2.2.3. Finally, (7) ensures that a query's workload shares on nodes k sum up to the shares assigned to the chunk, cf. $\bar{z}$. To minimize the worst-case workload share over all nodes and scenarios, in (6) we use a continuous variable L and add a penalty term $\alpha \cdot L$ in the objective (3). Hence, to achieve an even load balance, the parameter α has to be sufficiently large compared to K (e.g., $\alpha = 1\,000$); note, the factor W/V is bounded by K, cf. full replication.

The recursive decomposition principle of the LP (3) - (7) works as follows. Let x^*, y^*, and z^* denote the optimal solution of the LP (3) - (7) for a certain split. Then, for each subnode b, the input for its associated subproblem on the next level is characterized by the selected fragments (i.e., where $\bar{x}_{\bar{i}} := x^*_{i,b}$ is 1), the executable queries (i.e., where $\bar{y}_{\bar{j}} := y^*_{j,b}$ is 1), and the assigned workload shares (i.e., where $\bar{z}_{\bar{j},s} := z^*_{j,b,s}$ is positive). In this context, the top node represents the total workload, initially characterized by $\bar{x}_i := 1$, $\bar{y}_j := 1$, $\bar{z}_j := 1$, for all $i = 1, ..., N$ and $j = 1, ..., Q$. On the next decomposition level, the LP (3) - (7) is applied again for the new input. Recall, that for $B = K$ and $w_b = 1/K$, i.e., without using chunks, the LP guarantees an *optimal* allocation.

3.2 Heuristic Relaxation: Partial Clustering

The complexity of the LP (3) - (7) grows with the number of queries Q, fragments N, nodes K, and considered scenarios S. As a result, runtimes can get too large when considering huge workloads with thousands of queries and dozens of scenarios. While the decomposition heuristic allows dealing with larger numbers K, this does not solve the issue. As S can be chosen in a targeted way, the main limitation of our LP-based concept are large Q and N as they appear in real-world workloads (cf. $Q = 4\,461$, Section 2.3.2). Particularly Q is critical for the LP's complexity as the variables y and z as well as constraints (4), (5), and (7) are involved; instead, N is only relevant for x and does not affect the number of constraints. To still allow for short runtimes, we seek to address this problem by heuristically relaxing our LP.

The analysis of real-world workloads reveals, cf. Section 2.3, that the workload distribution of queries and accessed fragments is highly skewed (see Figure 1). Exploiting this property, we cluster queries and, in turn, simplify the complexity of the allocation problem as follows: (i) The majority of queries that represent only a small share of the workload are clustered within a set denoted by Q^F and assigned to the same node. (ii) The set of remaining (costly) queries denoted by, cf. (i),

$$Q^R := \{1, ..., Q\} \setminus Q^F \tag{8}$$

are used as (a smaller) input for the fragment allocation problem with K nodes, including the replica used for step (i). Following this concept, we propose the following approach.

Partial Clustering Approach: *Order the queries according to their expected loads over all scenarios $s = 1, ..., S$, cf. $E(f_{j,s}) \cdot c_j$ (if no distribution for s is given, we use uniform probabilities).* Then, assign the F queries with the smallest (expected) workload share, i.e., $Q^F := \{1, ..., F\}$, to one (e.g., the first) of the K nodes. Note, the number F has to be sufficiently small such that the workload share of the queries assigned to Q^F is (significantly) smaller than $1/K$. *The remaining workload $Q^R := \{F + 1, ..., Q\}$, cf. (8), has to be allocated to the other $K - 1$ nodes and the residual resources of the first cluster node ($k = 1$). The approach can be directly included in our LP model (3) - (7) via the additional constraints*

$$z_{j,1} = 1 \quad \forall j \in Q^F. \tag{9}$$

Note, the constraints (9) imply $y_{j,1} = 1$ for all $j \in Q^F$, cf. (5), as well as the allocation of all required fragments to the cluster node 1, cf. (4). If chunks are used, (9) is only active for the parent chunk $b = 1$ that is associated to the leaf node $k = 1$. Leaving some space on the cluster node allows the LP to assign other data-intensive queries to that node. Naturally, the LP's complexity decreases in F as we have fewer flexible queries ($Q - F$).

4 NUMERICAL EVALUATION

4.1 Results for a Single Fixed Workload

In this section, we compare the memory consumption and runtime of our partial clustering heuristic, cf. (3) - (9), against existing allocation approaches (see Section 2.2) for a single fixed workload scenario ($S = 1$) with $f_{j,1} := 1$, $j = 1, ..., Q$, for all queries.

4.1.1 TPC-DS Workload. For different numbers of nodes K, Table 2a summarizes the replication factor $\frac{W}{V}$ and runtime of our partial clustering heuristic (3) - (9). We used the penalty factor $\alpha := 1\,000$ and the Gurobi solver (version 9.0.0) (single-threaded).

The partial clustering approach allows reducing runtimes (by F via $|Q^F| = F$ and $|Q^R| = Q - F$) and is compatible with the decomposition approach. The results (Table 2a) show that our approach exploits both heuristics in a mutually supportive way. For instance, while for K=6 (TPC-DS), the optimal solution yields $W/V = 1.457$ in 1074 s, cp. Table 1a, our heuristic solution obtains $W/V = 1.584$ in 6 s. Compared to [12] (W^G) and [5] (W^D) we observe that via F we obtain a convenient mix of memory-efficiency and runtime, which has not been possible before.

4.1.2 Real-World Accounting Workload. We also applied our partial clustering approach for the real-world workload, cf. Section 2.3.2. For the example of $F = 4\,361$ fixed and $Q - F = 100$ flexibly assignable queries (being responsible for about 95% of the workload), Table 2b presents the results of our approach compared to the heuristics [5] and [12]. Compared to Table 1b, our partial clustering heuristic yields a competitive memory-efficiency (cf. $W/W^D < +7\%$) and runtimes below 10 s. Overall, we find that our approach can address both the performance limitations of [12] and the runtime limitations of [5].

Table 2: Best of both worlds: Memory-efficiency vs. runtime results achieved by partial clustering ($S = 1$): Our solution (W) with F fixed queries vs. [5] (W^D) and [12] (W^G).

(a) TPC-DS; $K = 4, ..., 12$, $N = 425$, $Q = 94$

K	F	chunks	$\frac{W}{V}$	solve time$_W$	$\frac{W}{W^D}$	$\frac{W}{W^G}$
4	36	4	1.314	1.3 s	+1.2%	-47.9%
5	47	5	1.501	2.0 s	+7.8%	-42.7%
6	4	3+3	1.584	5.5 s	+4.3%	-45.7%
8	15	4+4	1.957	2.6 s	+4.4%	-36.9%
10	47	5+5	2.330	5.0 s	+12.2%	-30.4%
12	15	4+4+4	2.416	7.0 s	+9.8%	-31.2%

(b) Real-world workload; $K = 4, ..., 12$, $N = 344$, $Q = 4\,461$, cf. source [1]

K	F	chunks	$\frac{W}{V}$	solve time$_W$	$\frac{W}{W^D}$	$\frac{W}{W^G}$
4	4 361	4	2.124	3.8 s	+ 0.9%	-41.9%
5	4 361	5	2.492	8.9 s	+ 0.8%	-35.8%
6	4 361	3+3	2.942	0.9 s	+ 3.0%	-35.1%
8	4 361	4+4	3.534	2.1 s	+ 1.0%	-45.9%
10	4 361	5+5	4.638	4.0 s	+ 3.9%	-40.2%
12	4 361	6+6	5.226	25.2 s	+ 1.2%	-44.5%
12	4 361	4+4+4	5.489	3.3 s	+ 6.3%	-41.7%

With decreasing F, the memory W decreases (improves), because the allocation gets more flexible. However, the runtime increases due to the larger remaining problem size. We observe that if the number of flexibly assigned queries $|Q^R|$ is large, the performance increase due to additional flexible queries is *diminishing* and does not justify the higher runtimes anymore.

Remark 1 *The partial clustering heuristic, cf. (3) - (9), is effective and flexible as it combines: (i) the reduction of problem complexity with workload knowledge and (ii) the possibility to exploit decomposition techniques (cf. Section 2.2.3). Combining these techniques allows balancing the solution quality and computation time in a targeted way such that memory-efficient solutions can be computed within short and plannable response times. Compared to the greedy heuristic (W^G), we find that the combined LP-based approach (W) requires (up to 48%) less data (within a similar computation time). Compared to the decomposition approach (W^G), we obtain solutions orders of magnitude (up to ×10 000) faster (using only slightly more memory).*

4.2 Multiple Workloads and Robustness

In this section, we compare fragment allocations for multiple scenarios, cf. $S > 1$. For workload scenario $s = 1$, we again let $f_{j,1} := 1$ for all queries j. For all other scenarios, we consider randomly generated query frequencies $f_{j,s}$ via $f_{j,s} :=$ if $U(0,1) < p$ then $1/p \cdot U(0,2)$ else 0, i.e., each query occurs only with probability p (cf. workloads with different queries, ad-hoc queries, etc.) and, on average, we have $E(f_{j,s}) = 1$, $j = 1, ..., Q$, $s = 2, ..., S$. In our examples, we use $p = 0.75$. Out-of-sample workloads, cf. $\tilde{s} = 1, ..., \tilde{S}$, for verification purposes are generated in the same way. Scenario-specific input details are available online [1].

4.2.1 TPC-DS Workload. Table 3a shows the results of our combined approach (3) - (9), cf. $W(S)$, for different numbers $F = 0, 15, 47, 62$ of fixed queries and different numbers of seen scenarios S (up to 100). While our approach without fixed queries ($F = 0$) is time-consuming, the partial clustering approach is again effective and allows deriving allocations for dozens of scenarios S quickly. We find that the required amount of data $\frac{W}{V}$ is overall (concave) increasing in S. Our replication factor is still significantly below K (cf. full replication) and (for the same S) far better than those of [12]'s merge approach, cf. $W^G(S)$, Section 2.5.

Moreover, we compared the *robustness* of our approach to the merge approach of [12] by analyzing the allocations' performance for *unseen* workloads. To calculate the worst (highest) workload share over all nodes (denoted by $\tilde{L}$) for a given scenario $\tilde{s}$, we can directly use the LP (3) - (7) without chunking ($B = K$) to evaluate a given fragment allocation $\vec{x}_{fix}$ by fixing the variables $\vec{x} := \vec{x}_{fix}$; (this also determines y, cf. (4)). When solving the LP for a *new* (randomly generated) workload scenario $\tilde{s}$, $\tilde{s} = 1, ..., \tilde{S}$, the remaining variables z and L are then chosen such that L is as small as possible and coincides with $\tilde{L}$. The *average* worst-case workload share over all $\tilde{S}$ unseen scenarios is denoted by $E(\tilde{L})$.

Table 3a shows the results for $\tilde{S} = 100$ unseen randomly generated workload scenarios. If more scenarios are taken into account, cf. S, the robustness improves, while the required amount of data and runtime increase. We observe that the average difference of $E(\tilde{L})$ and $1/K$ over all unseen (out-of-sample) scenarios is (concave) decreasing in the number of seen scenarios S. In our example, considering $S = 10$ (randomly chosen) scenarios were, on average, enough to obtain a fragment allocation that is robust against various unseen workloads by achieving an optimality gap for a node's highest workload share of $E^{(S)}(\tilde{L}) - 1/K \leq 0.0089$,

Table 3: Adding Robustness: Performance comparison with $\tilde{S} = 100$ random unseen workloads for different numbers seen workloads S. Memory vs. average out-of-sample workload limits $\tilde{L}$ of our partial clustering ($W(S)$) compared to the greedy merge approach [12] ($W^G(S)$).

(a) TPC-DS; $K = 8 = 4 + 4$, $N = 425$, $Q = 94$, F fixed queries, cf. (9)

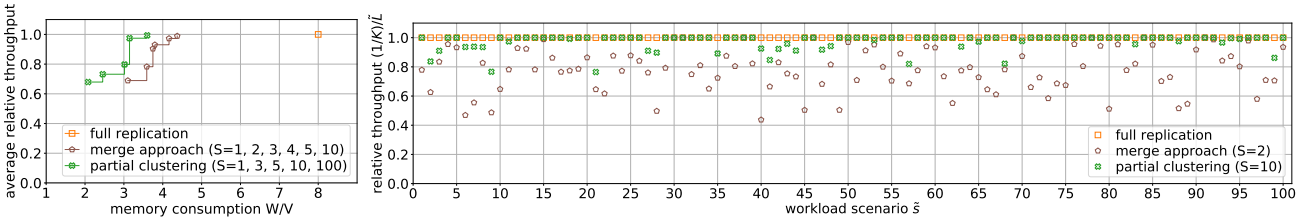
Approach	S	F	$\frac{W}{V}$	solve time	$E(\tilde{L}) - \frac{1}{K}$	$E\left(\frac{1/K}{\tilde{L}}\right)$
$W(S)$	1	0	1.874	110.5 s	0.0641	0.693
$W(S)$	3	0	2.208	175.7 s	0.0538	0.736
$W(S)$	5	0	2.702	345.6 s	0.0305	0.834
$W(S)$	7	0	2.756	286.0 s	0.0342	0.823
$W(S)$	10	0	2.721	561.8 s	0.0073	0.953
$W(S)$	1	47	2.079	1.8 s	0.0681	0.679
$W(S)$	3	47	2.455	2.9 s	0.0537	0.732
$W(S)$	5	47	3.016	2.9 s	0.0388	0.798
$W(S)$	7	47	3.057	6.3 s	0.0325	0.828
$W(S)$	10	15	2.958	42.0 s	0.0089	0.945
$W(S)$	10	47	3.145	20.2 s	0.0037	0.974
$W(S)$	20	47	3.212	11.9 s	0.0037	0.975
$W(S)$	50	47	3.275	36.7 s	0.0015	0.990
$W(S)$	100	47	3.600	331.5 s	0.0010	0.994
$W(S)$	100	62	4.039	112.5 s	0.0005	0.997
$W^G(S)$	1	/	3.101	<1 s	0.0654	0.689
$W^G(S)$	2	/	3.589	<1 s	0.0414	0.782
$W^G(S)$	3	/	3.747	<1 s	0.0155	0.904
$W^G(S)$	5	/	4.162	<1 s	0.0041	0.973
$W^G(S)$	10	/	4.372	<2 s	0.0017	0.989
$W^G(S)$	20	/	4.596	<3 s	0.0005	0.996
$W^G(S)$	50	/	5.528	<6 s	0.0000	1.000

(b) Real-world workload; $K = 8 = 4 + 4$, $N = 344$, $Q = 4\,461$, cf. source [1]

Approach	S	F	$\frac{W}{V}$	solve time	$E(\tilde{L}) - \frac{1}{K}$	$E\left(\frac{1/K}{\tilde{L}}\right)$
$W(S)$	1	4 361	3.534	2.1 s	0.0435	0.769
$W(S)$	3	4 361	3.906	2.8 s	0.0308	0.830
$W(S)$	5	4 361	3.953	6.8 s	0.0264	0.851
$W(S)$	10	4 361	5.008	7.7 s	0.0141	0.921
$W(S)$	10	4 411	5.593	3.8 s	0.0048	0.971
$W(S)$	20	4 361	5.505	8.7 s	0.0007	0.995
$W(S)$	50	4 361	5.743	28.4 s	0.0001	0.999
$W(S)$	100	4 361	5.847	93.7 s	0.0001	0.999
$W(S)$	100	4 411	6.183	23.7 s	0.0000	1.000
$W^G(S)$	1	/	6.536	<3 s	0.0040	0.971
$W^G(S)$	3	/	6.792	<10 s	0.0013	0.991
$W^G(S)$	5	/	6.913	<16 s	0.0013	0.991
$W^G(S)$	10	/	7.040	<34 s	0.0000	1.000

which is by far better than the standard $S = 1$ solution (cf. $E^{(S)}(\tilde{L}) - 1/K = 0.0641$ for $F = 0$) that optimizes only against the expected load. Naturally, the higher robustness with $S = 10$ is achieved by using more data, i.e., a higher replication factor of 2.721 (in 562 s, $F = 0$) and 3.145 (in 20 s, $F = 47$), instead of 1.874 ($F = 0$) and 2.079 ($F = 47$) when using only one scenario ($S = 1$). Note, compared to achieving robustness via full replication (with $W/V = K = 8$), this is a remarkable result.

Further, we evaluated the robustness of the merge approach, cf. $W^G(S)$, for the same ($\tilde{S}$) unseen workloads. For $S = 5$, we obtained the average worst-case workload limit $E^{(G)}(\tilde{L}) - 1/K = 0.0041$ and replication factor $W^G(S)/V = 4.162$. Compared to that, our $S = 10$ solution with $F = 47$ fixed queries obtains a *better robustness* (cf. $E^{(S)}(\tilde{L}) - 1/K = 0.0037$) and requires *clearly less memory* $W^G(S)/V = 3.145$. The better combinations of memory (W/V) and robustness can be observed over the full range of S. Figure 2a visualizes this result: For selected S, the figure shows the expected throughput (average over $\tilde{S} = 100$ unseen workloads)



(a) Memory consumption vs. average relative throughput over 100 unseen scenarios.

(b) Relative throughput for all 100 unseen scenarios.

Figure 2: Performance of allocation approaches for $\tilde{S} = 100$ unseen workload scenarios based on TPC-DS; $K = 8$.

per memory consumption for full replication, the merge approach, and our partial clustering. For merged allocations with $S = 2$ and our partial clustering with $S = 10$ input scenarios, Figure 2b shows the expected throughput $(1/K)/\tilde{L}$ for all $\tilde{S} = 100$ individual unseen scenarios.

Naturally, the specific results depend on underlying random numbers. However, using repeated simulations, we verified that the overall properties remain.

4.2.2 Real-World Accounting Workload. The results for the real-world workload (see Table 3b) show that our solution also remains applicable for larger workloads. The number of fixed queries F can be chosen such that for up to 20 scenarios, the runtime is below 10 s. A good trade-off (depending on the targets in practice) between robustness, runtime, and memory is achieved for 5 to 20 in-sample scenarios, cf. S.

Compared to the merge approach, cf. $W^G(S)$, we again obtain that our approach clearly outperforms their combinations of memory and throughput robustness against uncertain workloads. Moreover, our approach provides the flexibility to *tune* results by adjusting S and F according to a decision-maker's preferences.

Compared to TPC-DS, we find that, for all S, the optimality gap $E^{(S)}(\tilde{L}) - 1/K$ is on average lower, while replication factors are higher. This difference indicates that, in such cases, a *smaller* number of scenarios might be necessary to obtain a certain robustness. It can be explained by the fact that the workload is distributed over a higher number of queries Q, making the impact of single query frequencies, on average, less important.

Remark 2 *We find that optimizing the memory for an expected workload only is not robust against unseen workloads. Less memory-efficient approaches as [12] and particularly its merge extension are (indirectly) more robust against out-of-sample workloads as they systematically allocate more data. However, our approach to directly minimize required memory for multiple seen workloads, yields a better robustness using the same or even less data. The memory-efficiency of our LP-based approach allows including more scenarios within a certain memory budget than the merge approach [12]. Being able to deal with more representative workload scenarios, in turn, allows to better deal with altered unseen ones.*

5 RELATED WORK

Database replication is a means to improve availability and increase processing capabilities, and is supported in many systems, e.g., in HANA [10], Postgres-R [8], and as replication middleware [3]. Thereby, most systems implement full replication.

To calculate partial allocations, Rabl and Jacobsen propose a greedy algorithm (Section 2.2.2). For the same problem, we propose an LP-based decomposition approach [5] (Section 2.2.3). In both papers, the authors only consider a comparably small benchmark workload (TPC-H) and do neither (i) derive results for large-size workloads, (ii) study techniques to reduce the computation time for allocations, nor (iii) evaluate robustness against uncertain workloads. In [6], we visualize calculated allocations and investigate the intermediate steps taken by different algorithms. In [7], we visualize the load balancing behavior of allocations.

Özsu and Valduriez present a general overview of allocation problems in the field of distributed database systems [11]. They point out that allocation problems differ in constraints and optimization goals, such as performance, costs, and dependability.

Archer et al. [2] address an allocation problem, which is similar to ours. They evenly load-balance queries for web search containing multiple terms, which correspond to the fragments in our model. They cluster queries with a distributed balanced graph partitioning tool.

6 CONCLUSIONS

To overcome the limitations of existing allocation approaches [5, 12], we proposed a novel heuristic that combines different concepts. First, to achieve *memory-efficiency*, instead of rule-based heuristics, we leverage the power of LP techniques. Second, for *short calculation times*, we exploit that (real-world) workloads are skewed and reduce the problem complexity by a partial clustering approach that assigns a majority of queries with comparably small workload shares to a fixed node. Third, to add *robustness*, we force the allocation to be prepared against multiple diversified workloads. Moreover, we are able to balance the three target dimensions of the problem *flexibly*. Using TPC-DS and a large real-world workload, we verified the applicability and effectiveness of our approach, which clearly outperformed existing approaches regarding the mix of the three key dimensions.

REFERENCES

[1] [n.d.]. GitHub repository containing the workload data and source code. https://hyrise.github.io/replication/.
[2] Aaron Archer et al. 2019. Cache-aware load balancing of data center applications. *PVLDB* 12, 6 (2019), 709–723.
[3] Emmanuel Cecchet et al. 2008. Middleware-based database replication: the gaps between theory and practice. In *SIGMOD*. 739–752.
[4] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. 1996. The Dangers of Replication and a Solution. In *SIGMOD*. 173–182.
[5] Stefan Halfpap et al. 2019. Workload-Driven Fragment Allocation for Partially Replicated Databases Using Linear Programming. In *ICDE*. 1746–1749.
[6] Stefan Halfpap and Rainer Schlosser. 2019. A Comparison of Allocation Algorithms for Partially Replicated Databases. In *ICDE*. 2008–2011.
[7] Stefan Halfpap and Rainer Schlosser. 2020. Exploration of Dynamic Query-Based Load Balancing for Partially Replicated Database Systems with Node Failures. In *CIKM*. 3409–3412.
[8] Bettina Kemme et al. 2000. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *VLDB*. 134–143.
[9] Jens Krüger et al. 2011. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB* 5, 1 (2011), 61–72.
[10] Juchang Lee et al. 2017. Parallel Replication across Formats in SAP HANA for Scaling Out Mixed OLTP/OLAP Workloads. *PVLDB* 10, 12 (2017), 1598–1609.
[11] M. Tamer Özsu and Patrick Valduriez. 2011. *Principles of Distributed Database Systems, Third Edition*. Springer.
[12] Tilmann Rabl and Hans-Arno Jacobsen. 2017. Query Centric Partitioning and Allocation for Partially Replicated Database Systems. In *SIGMOD*. 315–330.

CONTENTS

Robust and Memory-Efficient Database Fragment Allocation for Large and Uncertain Database Workloads

Rainer Schlosser*
Hasso Plattner Institute, Potsdam, Germany
rainer.schlosser@hpi.de

Stefan Halfpap*
Hasso Plattner Institute, Potsdam, Germany
stefan.halfpap@hpi.de

ABSTRACT

Database replication and query load-balancing are mechanisms to scale query throughput. The analysis of workloads allows load-balancing queries to replica nodes according to their accessed data. As a result, replica nodes must only store and synchronize subsets of the data. However, balancing the load of large-scale workloads evenly while minimizing the memory footprint is complex and challenging. State-of-the-art allocation approaches are either time consuming or the resulting allocations are not memory-efficient. Further, partial replication approaches usually optimize only against a single fixed workload. If the actual workload deviates from this expected one, load-balancing can be highly skewed, resulting in severe performance degradation.

This paper proposes a novel approach to compute memory-efficient fragment allocations that enable balancing multiple potential future workloads. Applied on the TPS-DS benchmark and a large-size enterprise workload, we show that, compared to state-of-the-art allocations, our solutions are (i) more flexible, (ii) require up to 50% less data, (iii) have competitive runtimes, and (iv) are more robust against uncertain out-of-sample workloads.

1 INTRODUCTION

Increasing demand for database processing capabilities can be managed by scale-out approaches, using additional servers. Analyses of enterprise workloads have shown that both OLTP and OLAP are read-dominant [9]. Database replication is an approach to scale-out read-only queries, which can be executed on snapshots of a primary server without violating consistency [4].

Given that most queries require only a limited set of tuples and attributes, partial replication is an efficient approach: Instead of duplicating all data to all replica nodes, partial replicas store only a subset of the data while being able to process a large workload share. Thereby, splitting the overall workload evenly among the replica nodes is essential to scale the query throughput linearly with the number of nodes. Partial replication consists of two steps, which are typically separated from each other to better deal with the problem complexity [11]. First, the data set is partitioned horizontally and/or vertically into disjoint data partitions/fragments. Second, the individual fragments are allocated to one or multiple nodes.

This paper addresses the second step, i.e., a fragment allocation problem (given a fixed data partitioning). Calculating efficient fragment allocations that minimize the replicas' memory consumption while evenly balancing the query load is challenging, because the data allocation and the workload distribution are mutually dependent. As the calculation of optimal allocations is an NP-hard problem, heuristic approaches have to be used for large problem sizes. Rabl and Jacobsen propose a greedy allocation approach with short computation times [12]. We have previously proposed a decomposition approach [5] based on linear programming (LP), which calculates allocations with up to 23% lower memory consumption for the TPC-H benchmark.

Allocations for larger workloads are harder to solve, but typically offer greater potential for sophisticated approaches compared to simple heuristics. However, when using the LP-based decomposition approach for larger problems, computation times increase, and problems may finally become practically intractable, e.g., for an application in dynamic settings, in which model inputs change and quick recalculations are required. Considering multiple potential workloads to increase an allocation's robustness increases the problem complexity even further. However, such robustness is necessary in practice, when workloads fluctuate, and query costs or frequencies cannot be predicted precisely.

The goal of this paper is to overcome the limitations of existing allocation approaches. Applied to TPC-DS and a real-world accounting workload, we show that the greedy rule-based heuristic [12] is not memory-efficient, while the solver-based solution [5] provides low robustness against diversified workloads and has unacceptable runtimes to be used in practice. To fill this gap, we propose a heuristic LP-based clustering approach to *flexibly combine robustness, memory-efficiency, and a short calculation time* for large-scale problems. Our *contributions* are the following:

First, we derive robust and memory-efficient fragment allocations, which enable an even load balancing against multiple potential future workloads. Second, exploiting the skewness of workloads, we use partial clustering techniques to compute solutions for large-scale workloads quickly. Third, for the TPC-DS and a large real-world workload, we show that, with our techniques, the trade-off between memory-efficiency, robustness, and a short calculation time can be smoothly balanced in a targeted way. Fourth, we verify the robustness of our allocations by confronting them with unseen out-of-sample workloads.

2 FRAGMENT ALLOCATION PROBLEM AND LIMITS OF EXISTING APPROACHES

2.1 Problem Description and Difficulty

The scale-out of workloads to partially replicated databases leads to a coupled data assignment and workload distribution problem. We consider a horizontally and/or vertically partitioned database consisting of N disjoint fragments.

The workload input can be described as follows. We consider the case where data (fragments) can be stored on K nodes to distribute a given workload. The size of a fragment i is a_i, $i = 1, ..., N$. Further, we consider a set of Q queries j, characterized by fragments accessed, i.e., $q_j \subseteq \{1, ..., N\}$, $j = 1, ..., Q$. We assume that the costs of queries j are independent of the executing node k, $k = 1, ..., K$, and determined by c_j, $j = 1, ..., Q$. Query costs are numerical and can, e.g., be modeled as average processing times. We assume that the queries j occur with a given frequency f_j, $j = 1, ..., Q$. The total workload costs C are $C := \sum_{j=1,...,Q} f_j \cdot c_j$.

*Both authors contributed equally to this research.

The allocation problem can be described as follows. The goal is to decide (i) on which node to put which fragments and (ii) which query is executed at which node to which extent (workload share). Our objective is to *minimize* the total amount of data at all nodes such that the workload can be evenly distributed between them. Further: (i) A query j can only be executed at node k if all relevant fragments are stored on node k. (ii) For each of the Q queries, the workload shares, assigned to the different nodes, have to sum up to one. (iii) At each of the K nodes, the workload share has to be $1/K$ such that the overall query throughput can be scaled. (iv) Further, as high calculation times may limit the applicability in practice, we want to compute optimized allocations quickly. (v) The allocation should work for multiple given workload scenarios and should be robust against new unseen ones.

2.2 Existing Fragment Allocation Approaches

2.2.1 Optimal Solution via Linear Programming. For a single given workload, the described fragment allocation problem can be formulated as a linear mixed integer problem, cf. [5, 12]. However, the complexity of the linear program quickly increases with the number of queries (Q), fragments (N), and nodes (K). For this reason, the optimal solution can only be derived as long as the size of the problem is sufficiently *small*.

2.2.2 Greedy Heuristic. Rabl and Jacobsen [12] start to assign queries with the largest workload share and accessing the most data. Queries are ordered by the product of the workload share and the total size of accessed fragments. A query is assigned to the node with the largest overlap of already allocated fragments and those accessed by the query. Nodes with no assigned queries are thereby treated as if they have a complete overlap. If a query's workload share exceeds the assigned node's load capacity, the node is filled up to its limit. The query with its remaining workload is merged back into the list of queries and assigned later.

2.2.3 LP-Based Decomposition. We proposed to iteratively *split* the workload into smaller workload packages (chunks) such that the data redundancy is minimized in each step [5]. In a split with B subnodes, each subnode b represents n_b final nodes, $b = 1, ..., B$, and takes the workload share $w_b := n_b/K$. The special case $B = K$ corresponds to the optimal solution, cf. Section 2.2.1. The workload splits are obtained using small-sized LP *subproblems* similar to the LP structure of optimal solutions. In the LP, the following variables are used: The binary variables $x_{i,k} \in \{0, 1\}$, $i = 1, ..., N$, $k = 1, ..., K$, indicate whether fragment i is allocated to node k (1) or not (0). The binary variables $y_{j,k} \in \{0, 1\}$, $j = 1, ..., Q$, $k = 1, ..., K$, indicate whether query j can run on node k (1) or not (0). The continuous variables $z_{j,k} \in [0, 1]$, $j = 1, ..., Q$, $k = 1, ..., K$, represent the workload share of query j executed at node k. The sum of shares has to sum up to one for all queries. Further, by W/V, the *replication factor* is denoted, where the total amount of data used

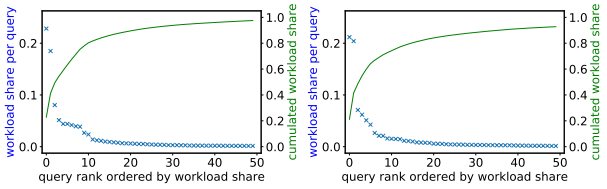
$$W := \sum_{i=1,..,N,k=1,...,K} x_{i,k} \cdot a_i \tag{1}$$

is normalized by the amount of overall accessed data

$$V := \sum_{i \in \bigcup_{j=1,...,Q: f_j > 0} \{q_j\}} a_i. \tag{2}$$

2.3 Large and Skewed Workloads

The results of [5] and [12] were shown for the TPC-H benchmark, which consists of $Q = 22$ queries and $N = 61$ fragments/columns. We use the more complex TPC-DS benchmark ($Q = 99$, $N = 425$) and a real-world enterprise workload ($Q = 4\,461$, $N = 344$).



(a) TPC-DS **(b) Accounting workload**

Figure 1: Distribution of top 50 query workload shares in decreasing order, cf. (a) Section 2.3.1 and (b) Section 2.3.2.

2.3.1 TPC-DS Workload. To obtain model inputs, we loaded the tables with scale factor 1 into a PostgreSQL 12.2 database system. We use vertical partitioning with each column as an individual fragment. We deployed single column indices on all primary key columns. Fragment sizes a_i, $i = 1, ..., 425$, are modeled by using the function `pg_column_size()` to calculate the pure value sizes, abstracting from the PostgreSQL page layout with meta-information and padding. In case the column is part of a primary key, the index size increases the associated fragment size. We use the command `pg_table_size(index_name)` to calculate index sizes. We derived query costs c_j as average execution time for query template j with varying parameters. For TPC-DS queries 1, 4, 6, 11, and 74, the set timeout of 120 s was exceeded. Thus, we omitted them in our experiments, resulting in $Q = 94$ queries.

2.3.2 Real-World Accounting Workload. We got access to metadata of an enterprise's central accounting table and a summary of a workload trace against this table in the form of query templates and statistics. The metadata enabled us to derive all required model inputs for calculating fragment allocations using vertical partitioning. The anonymized workload metadata and source code to reproduce the allocations are publicly available online [1]. The analyzed table stores accounting information and has N=344 columns. The summary of the workload trace consists of Q=4 461 SQL templates with aggregated execution properties of individual queries. Thereby, the most important properties for our research are query frequencies f_j (occurrences) and costs c_j, i.e., the average execution time per query (template).

2.3.3 Workload Skewness. Figure 1 shows the distribution of and cumulative query workload shares $f_j \cdot c_j$, $j = 1, ..., Q$. For both workloads, the distribution is *highly skewed*: The queries with the 50 highest workload shares account for more than 97% of the TPC-DS and more than 92% of the real-world workload.

2.4 Limitations of Existing Approaches

To study the suitability of existing approaches, we calculated allocations for TPC-DS and a real-world workload, cf. Section 2.3. For different numbers K, Table 1 summarizes memory consumption and runtime of existing allocations approaches. We used $f_j := 1$, for all $j = 1, ..., Q$. The LP-results of [5] were achieved using the Gurobi solver (version 9.0.0) (single-threaded). For the comparison with [12]'s approach, we implemented their algorithms in Python 3, and declare the runtime as an upper limit (<).

The upper part of both subtables shows the memory consumption (W/V) and required runtime for the optimal solution, cf. W^D without chunking, for up to $K = 6$ (TPC-DS) and $K = 5$ (accounting workload) nodes compared to the greedy heuristic, cf. W^G. We could not compute optimal allocations for larger numbers of nodes K within 8 hours. The optimal replication factors provide a useful reference to verify the quality of heuristic approaches. In this context, we observe that the greedy heuristic requires up to 100% more data than the optimal solution.

Table 1: Advantages and disadvantages of existing approaches: the greedy heuristic [12] (W^G), cf. Sec. 2.2.2, vs. the decomposition heuristic [5] (W^D), cf. Sec. 2.2.3, including optimal solutions, cf. Sec. 2.2.1, (*no decomposition).

(a) TPC-DS; $K = 2, ..., 12$, $N = 425$, $Q = 94$.

K	chunks	$\frac{W^D}{V}$	solve time$_{W^D}$	$\frac{W^G}{W^D}$	solve time$_{W^G}$
2	2	1.126*	1 s	+1%	<0.1 s
3	3	1.205*	9 s	+53%	<0.1 s
4	4	1.298*	43 s	+94%	<0.1 s
5	5	1.393*	407 s	+88%	<0.1 s
6	6	1.457*	1074 s	+100%	<0.1 s
4	2+2	1.310	2 s	+93%	<0.1 s
5	3+2	1.466	14 s	+79%	<0.1 s
6	3+3	1.519	14 s	+92%	<0.1 s
8	4+4	1.874	122 s	+65%	<0.1 s
10	5+5	2.076	517 s	+61%	<0.1 s
12	6+6	2.201	2 146 s	+60%	<0.1 s

(b) Real-world workload; $K = 2, ..., 12$, $N = 344$, $Q = 4\,461$, cf. source [1].

K	chunks	$\frac{W^D}{V}$	solve time$_{W^D}$	$\frac{W^G}{W^D}$	solve time$_{W^G}$
2	2	1.322*	179 s	+51%	<3 s
3	3	1.775*	6 236 s	+50%	<3 s
4	4	2.104*	9 356 s	+74%	<3 s
5	5	2.473*	13 738 s	+57%	<3 s
3	2+1	1.811	384 s	+47%	<3 s
4	2+2	2.126	225 s	+72%	<3 s
5	2+2+1	2.499	5 922 s	+55%	<3 s
6	3+3	2.855	778 s	+59%	<3 s
8	3+3+2	3.499	8 859 s	+87%	<3 s
10	4+3+3	4.462	49 233 s	+74%	<3 s
12	4+4+4	5.162	47 207 s	+82%	<3 s

The lower part of the subtables shows the results of the decomposition heuristic compared to the greedy heuristic. The decomposition and greedy approach make it possible to solve the problem for larger K heuristically. We observe that the decomposition approach (W^D) yields better replication factors than the greedy approach (W^G), which requires up to 93% and 87% more data for TPC-DS and the accounting workload, respectively. Further, the decomposition approach performs close to optimal compared to the optimal solution results (for $K \leq 6$, see Table 1a and for $K \leq 5$, see Table 1b). However, the decomposition approach requires high computation times if the problem becomes large, e.g., in the case of the accounting workload. In contrast, the greedy approach is fast and requires only seconds.

2.5 Uncertain Future Workloads

In general, future workloads are not entirely predictable. Thus, a further weakness of existing approaches is that they are only optimized for a single workload and can perform poorly if actual workloads differ. Hence, it is crucial to take potential workload scenarios into account to obtain a robust performance. Potential future workload scenarios can be determined, e.g., based on previously observed (seasonal) workloads or forecasts. In this context, fragment allocations should be such that the workload can be successfully balanced in any scenario. We assume that a workload scenario is characterized by a set of queries with given frequencies and costs within a certain time span.

In [12], Rabl and Jacobsen also describe an extension of their approach to cope with multiple workload scenarios. They propose to calculate a separate allocation for each scenario independently. Individual allocations are merged pairwise, mapping each node of the first allocation to a node of the second allocation. A merged allocation enables an even load balancing for both input allocations. The Hungarian method allows calculating an optimal mapping, which minimizes the memory consumption of the merged allocation (in polynomial time). However, because entire nodes are merged, optimization potential is given away.

Overall, we observe that existing approaches have different strengths and weaknesses (runtime vs. memory-efficiency). Further, from a practical perspective, solutions are required that can provide a *reasonable combination* of (i) memory-efficiency, (ii) robustness against different workloads, and (iii) short runtimes. Our goal is to design a heuristic approach that is of that kind.

3 ROBUST FRAGMENT ALLOCATION FOR MULTIPLE POTENTIAL WORKLOADS

3.1 LP-Based Robust Solution Approach

In the following, we consider S potential workload scenarios. Each scenario s, $s = 1, ..., S$, is characterized by query frequencies $f_{j,s}$ and associated workload costs $C_s := \sum_{j=1,...,Q} f_{j,s} \cdot c_j$, cp. Section 2.1. Note, in this framework, also uncertain query costs c_j can be expressed similarly by using potential scenario-based costs $c_{j,s}$ without increasing the model's complexity.

The core idea is finding a single allocation that enables an even load balancing for all S potential workloads scenarios. Further, by enabling an even load balance for specific diversified scenarios, such enriched allocations also allow improved load balancing for *unseen* scenarios, which may be similar to mixtures of input scenarios. Thereby, an allocation's robustness can be increased by choosing a larger number of diverse scenarios.

Our robust multi-scenario model is an extension of the LP-based decomposition approach, cf. Section 2.2.3, which considers one deterministic workload, cf. $S = 1$. Compared to [5], we use extended variables (cf. z), to model workload shares for each scenario s, $s = 1, ..., S$. Based on the decomposition concept of [5], we propose the following extended LP model to allocate data fragments and to distribute workload shares to multiple nodes in the presence of multiple potential workloads:

$$\begin{aligned} &\textit{minimize} \\ &x_{i,b}, y_{j,b} \in \{0,1\}, z_{j,b,s} \in [0,1], 0 \leq L \leq 1, (\text{given } \bar{x}_i, \bar{y}_j, \bar{z}_{j,s}) \\ &i = 1, ..., N,\ j = 1, ..., Q,\ b = 1, ..., B,\ s = 1, ..., S \end{aligned}$$

$$\frac{1}{V} \cdot \sum_{i=1,..,N, b=1,...,B: \bar{x}_i=1} x_{i,b} \cdot a_i + \alpha \cdot L \tag{3}$$

$$s.t. \quad y_{j,b} \cdot |q_j| \leq \sum_{i \in q_j} x_{i,b}, \qquad \begin{matrix} j = 1, ..., Q : \bar{y}_j = 1 \\ b = 1, ..., B \end{matrix} \tag{4}$$

$$z_{j,b,s} \leq y_{j,b}, \qquad \begin{matrix} j = 1, ..., Q : \bar{y}_j = 1 \\ b = 1, ..., B, \quad s = 1, ..., S \end{matrix} \tag{5}$$

$$\sum_{j=1,...,Q: \bar{y}_j=1} f_{j,s} \cdot c_j / C_s / w_b \cdot z_{j,b,s} \leq L, \qquad \begin{matrix} b = 1, ..., B \\ s = 1, ..., S \end{matrix} \tag{6}$$

$$\sum_{b=1,...,B} z_{j,b,s} = \bar{z}_{j,s}, \qquad \begin{matrix} j = 1, ..., Q : \bar{y}_j = 1 \\ s = 1, ..., S \end{matrix} \tag{7}$$

The overall idea is that the model allocates the fragments such that the workload can be distributed evenly for all workload scenarios s, $s = 1, ..., S$. Note, scenario probabilities are not used and thus do not have to be quantified in advance.

The objective (3) minimizes the replication factor W/V, cp. (1) - (2). Constraint (4) guarantees that a query j can only be executed on node b if all relevant fragments are available, see Section 2.1. The cardinality term $|q_j|$ expresses the number of fragments used in query j. Constraint (5) ensures that a query j can only have a positive workload share on node b in scenario s if it can be executed on node b. Constraint (6) guarantees that,

in all scenarios s, all nodes b do not exceed the workload limit L. Here the workload is normalized by the total workload cost C_s of scenario s and the workload share $w_b := n_b/K$, cf. Sec. 2.2.3. Finally, (7) ensures that a query's workload shares on nodes k sum up to the shares assigned to the chunk, cf. $\bar{z}$. To minimize the worst-case workload share over all nodes and scenarios, in (6) we use a continuous variable L and add a penalty term $\alpha \cdot L$ in the objective (3). Hence, to achieve an even load balance, the parameter α has to be sufficiently large compared to K (e.g., $\alpha = 1\,000$); note, the factor W/V is bounded by K, cf. full replication.

The recursive decomposition principle of the LP (3) - (7) works as follows. Let x^*, y^*, and z^* denote the optimal solution of the LP (3) - (7) for a certain split. Then, for each subnode b, the input for its associated subproblem on the next level is characterized by the selected fragments (i.e., where $\bar{x}_{\bar{i}} := x^*_{i,b}$ is 1), the executable queries (i.e., where $\bar{y}_{\bar{j}} := y^*_{j,b}$ is 1), and the assigned workload shares (i.e., where $\bar{z}_{\bar{j},s} := z^*_{j,b,s}$ is positive). In this context, the top node represents the total workload, initially characterized by $\bar{x}_i := 1$, $\bar{y}_j := 1$, $\bar{z}_j := 1$, for all $i = 1, ..., N$ and $j = 1, ..., Q$. On the next decomposition level, the LP (3) - (7) is applied again for the new input. Recall, that for $B = K$ and $w_b = 1/K$, i.e., without using chunks, the LP guarantees an *optimal* allocation.

3.2 Heuristic Relaxation: Partial Clustering

The complexity of the LP (3) - (7) grows with the number of queries Q, fragments N, nodes K, and considered scenarios S. As a result, runtimes can get too large when considering huge workloads with thousands of queries and dozens of scenarios. While the decomposition heuristic allows dealing with larger numbers K, this does not solve the issue. As S can be chosen in a targeted way, the main limitation of our LP-based concept are large Q and N as they appear in real-world workloads (cf. $Q = 4\,461$, Section 2.3.2). Particularly Q is critical for the LP's complexity as the variables y and z as well as constraints (4), (5), and (7) are involved; instead, N is only relevant for x and does not affect the number of constraints. To still allow for short runtimes, we seek to address this problem by heuristically relaxing our LP.

The analysis of real-world workloads reveals, cf. Section 2.3, that the workload distribution of queries and accessed fragments is highly skewed (see Figure 1). Exploiting this property, we cluster queries and, in turn, simplify the complexity of the allocation problem as follows: (i) The majority of queries that represent only a small share of the workload are clustered within a set denoted by Q^F and assigned to the same node. (ii) The set of remaining (costly) queries denoted by, cf. (i),

$$Q^R := \{1, ..., Q\} \setminus Q^F \tag{8}$$

are used as (a smaller) input for the fragment allocation problem with K nodes, including the replica used for step (i). Following this concept, we propose the following approach.

Partial Clustering Approach: *Order the queries according to their expected loads over all scenarios $s = 1, ..., S$, cf. $E(f_{j,s}) \cdot c_j$ (if no distribution for s is given, we use uniform probabilities).* Then, assign the F queries with the smallest (expected) workload share, i.e., $Q^F := \{1, ..., F\}$, to one (e.g., the first) of the K nodes. Note, the number F has to be sufficiently small such that the workload share of the queries assigned to Q^F is (significantly) smaller than $1/K$. *The remaining workload $Q^R := \{F + 1, ..., Q\}$, cf. (8), has to be allocated to the other $K - 1$ nodes and the residual resources of the first cluster node ($k = 1$). The approach can be directly included in our LP model (3) - (7) via the additional constraints*

$$z_{j,1} = 1 \quad \forall j \in Q^F. \tag{9}$$

Note, the constraints (9) imply $y_{j,1} = 1$ for all $j \in Q^F$, cf. (5), as well as the allocation of all required fragments to the cluster node 1, cf. (4). If chunks are used, (9) is only active for the parent chunk $b = 1$ that is associated to the leaf node $k = 1$. Leaving some space on the cluster node allows the LP to assign other data-intensive queries to that node. Naturally, the LP's complexity decreases in F as we have fewer flexible queries ($Q - F$).

4 NUMERICAL EVALUATION

4.1 Results for a Single Fixed Workload

In this section, we compare the memory consumption and runtime of our partial clustering heuristic, cf. (3) - (9), against existing allocation approaches (see Section 2.2) for a single fixed workload scenario ($S = 1$) with $f_{j,1} := 1$, $j = 1, ..., Q$, for all queries.

4.1.1 TPC-DS Workload. For different numbers of nodes K, Table 2a summarizes the replication factor $\frac{W}{V}$ and runtime of our partial clustering heuristic (3) - (9). We used the penalty factor $\alpha := 1\,000$ and the Gurobi solver (version 9.0.0) (single-threaded).

The partial clustering approach allows reducing runtimes (by F via $|Q^F| = F$ and $|Q^R| = Q - F$) and is compatible with the decomposition approach. The results (Table 2a) show that our approach exploits both heuristics in a mutually supportive way. For instance, while for K=6 (TPC-DS), the optimal solution yields $W/V = 1.457$ in 1074 s, cp. Table 1a, our heuristic solution obtains $W/V = 1.584$ in 6 s. Compared to [12] (W^G) and [5] (W^D) we observe that via F we obtain a convenient mix of memory-efficiency and runtime, which has not been possible before.

4.1.2 Real-World Accounting Workload. We also applied our partial clustering approach for the real-world workload, cf. Section 2.3.2. For the example of $F = 4\,361$ fixed and $Q - F = 100$ flexibly assignable queries (being responsible for about 95% of the workload), Table 2b presents the results of our approach compared to the heuristics [5] and [12]. Compared to Table 1b, our partial clustering heuristic yields a competitive memory-efficiency (cf. $W/W^D < +7\%$) and runtimes below 10 s. Overall, we find that our approach can address both the performance limitations of [12] and the runtime limitations of [5].

Table 2: Best of both worlds: Memory-efficiency vs. runtime results achieved by partial clustering ($S = 1$): Our solution (W) with F fixed queries vs. [5] (W^D) and [12] (W^G).

(a) TPC-DS; $K = 4, ..., 12$, $N = 425$, $Q = 94$

K	F	chunks	$\frac{W}{V}$	solve time$_W$	$\frac{W}{W^D}$	$\frac{W}{W^G}$
4	36	4	1.314	1.3 s	+1.2%	-47.9%
5	47	5	1.501	2.0 s	+7.8%	-42.7%
6	4	3+3	1.584	5.5 s	+4.3%	-45.7%
8	15	4+4	1.957	2.6 s	+4.4%	-36.9%
10	47	5+5	2.330	5.0 s	+12.2%	-30.4%
12	15	4+4+4	2.416	7.0 s	+9.8%	-31.2%

(b) Real-world workload; $K = 4, ..., 12$, $N = 344$, $Q = 4\,461$, cf. source [1]

K	F	chunks	$\frac{W}{V}$	solve time$_W$	$\frac{W}{W^D}$	$\frac{W}{W^G}$
4	4 361	4	2.124	3.8 s	+ 0.9%	-41.9%
5	4 361	5	2.492	8.9 s	+ 0.8%	-35.8%
6	4 361	3+3	2.942	0.9 s	+ 3.0%	-35.1%
8	4 361	4+4	3.534	2.1 s	+ 1.0%	-45.9%
10	4 361	5+5	4.638	4.0 s	+ 3.9%	-40.2%
12	4 361	6+6	5.226	25.2 s	+ 1.2%	-44.5%
12	4 361	4+4+4	5.489	3.3 s	+ 6.3%	-41.7%

With decreasing F, the memory W decreases (improves), because the allocation gets more flexible. However, the runtime increases due to the larger remaining problem size. We observe that if the number of flexibly assigned queries $|Q^R|$ is large, the performance increase due to additional flexible queries is *diminishing* and does not justify the higher runtimes anymore.

Remark 1 *The partial clustering heuristic, cf. (3) - (9), is effective and flexible as it combines: (i) the reduction of problem complexity with workload knowledge and (ii) the possibility to exploit decomposition techniques (cf. Section 2.2.3). Combining these techniques allows balancing the solution quality and computation time in a targeted way such that memory-efficient solutions can be computed within short and plannable response times. Compared to the greedy heuristic (W^G), we find that the combined LP-based approach (W) requires (up to 48%) less data (within a similar computation time). Compared to the decomposition approach (W^G), we obtain solutions orders of magnitude (up to ×10 000) faster (using only slightly more memory).*

4.2 Multiple Workloads and Robustness

In this section, we compare fragment allocations for multiple scenarios, cf. $S > 1$. For workload scenario $s = 1$, we again let $f_{j,1} := 1$ for all queries j. For all other scenarios, we consider randomly generated query frequencies $f_{j,s}$ via $f_{j,s} :=$ if $U(0,1) < p$ then $1/p \cdot U(0,2)$ else 0, i.e., each query occurs only with probability p (cf. workloads with different queries, ad-hoc queries, etc.) and, on average, we have $E(f_{j,s}) = 1$, $j = 1, ..., Q$, $s = 2, ..., S$. In our examples, we use $p = 0.75$. Out-of-sample workloads, cf. $\tilde{s} = 1, ..., \tilde{S}$, for verification purposes are generated in the same way. Scenario-specific input details are available online [1].

4.2.1 TPC-DS Workload. Table 3a shows the results of our combined approach (3) - (9), cf. $W(S)$, for different numbers $F = 0, 15, 47, 62$ of fixed queries and different numbers of seen scenarios S (up to 100). While our approach without fixed queries ($F = 0$) is time-consuming, the partial clustering approach is again effective and allows deriving allocations for dozens of scenarios S quickly. We find that the required amount of data $\frac{W}{V}$ is overall (concave) increasing in S. Our replication factor is still significantly below K (cf. full replication) and (for the same S) far better than those of [12]'s merge approach, cf. $W^G(S)$, Section 2.5.

Moreover, we compared the *robustness* of our approach to the merge approach of [12] by analyzing the allocations' performance for *unseen* workloads. To calculate the worst (highest) workload share over all nodes (denoted by $\tilde{L}$) for a given scenario $\tilde{s}$, we can directly use the LP (3) - (7) without chunking ($B = K$) to evaluate a given fragment allocation $\vec{x}_{fix}$ by fixing the variables $\vec{x} := \vec{x}_{fix}$; (this also determines y, cf. (4)). When solving the LP for a *new* (randomly generated) workload scenario $\tilde{s}$, $\tilde{s} = 1, ..., \tilde{S}$, the remaining variables z and L are then chosen such that L is as small as possible and coincides with $\tilde{L}$. The *average* worst-case workload share over all $\tilde{S}$ unseen scenarios is denoted by $E(\tilde{L})$.

Table 3a shows the results for $\tilde{S} = 100$ unseen randomly generated workload scenarios. If more scenarios are taken into account, cf. S, the robustness improves, while the required amount of data and runtime increase. We observe that the average difference of $E(\tilde{L})$ and $1/K$ over all unseen (out-of-sample) scenarios is (concave) decreasing in the number of seen scenarios S. In our example, considering $S = 10$ (randomly chosen) scenarios were, on average, enough to obtain a fragment allocation that is robust against various unseen workloads by achieving an optimality gap for a node's highest workload share of $E^{(S)}(\tilde{L}) - 1/K \leq 0.0089$,

Table 3: Adding Robustness: Performance comparison with $\tilde{S} = 100$ random unseen workloads for different numbers seen workloads S. Memory vs. average out-of-sample workload limits $\tilde{L}$ of our partial clustering ($W(S)$) compared to the greedy merge approach [12] ($W^G(S)$).

(a) TPC-DS; $K = 8 = 4 + 4$, $N = 425$, $Q = 94$, F fixed queries, cf. (9)

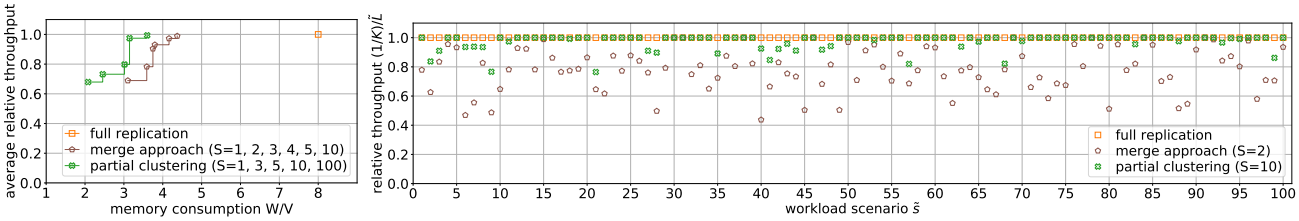
Approach	S	F	$\frac{W}{V}$	solve time	$E(\tilde{L}) - \frac{1}{K}$	$E\left(\frac{1/K}{\tilde{L}}\right)$
$W(S)$	1	0	1.874	110.5 s	0.0641	0.693
$W(S)$	3	0	2.208	175.7 s	0.0538	0.736
$W(S)$	5	0	2.702	345.6 s	0.0305	0.834
$W(S)$	7	0	2.756	286.0 s	0.0342	0.823
$W(S)$	10	0	2.721	561.8 s	0.0073	0.953
$W(S)$	1	47	2.079	1.8 s	0.0681	0.679
$W(S)$	3	47	2.455	2.9 s	0.0537	0.732
$W(S)$	5	47	3.016	2.9 s	0.0388	0.798
$W(S)$	7	47	3.057	6.3 s	0.0325	0.828
$W(S)$	10	15	2.958	42.0 s	0.0089	0.945
$W(S)$	10	47	3.145	20.2 s	0.0037	0.974
$W(S)$	20	47	3.212	11.9 s	0.0037	0.975
$W(S)$	50	47	3.275	36.7 s	0.0015	0.990
$W(S)$	100	47	3.600	331.5 s	0.0010	0.994
$W(S)$	100	62	4.039	112.5 s	0.0005	0.997
$W^G(S)$	1	/	3.101	<1 s	0.0654	0.689
$W^G(S)$	2	/	3.589	<1 s	0.0414	0.782
$W^G(S)$	3	/	3.747	<1 s	0.0155	0.904
$W^G(S)$	5	/	4.162	<1 s	0.0041	0.973
$W^G(S)$	10	/	4.372	<2 s	0.0017	0.989
$W^G(S)$	20	/	4.596	<3 s	0.0005	0.996
$W^G(S)$	50	/	5.528	<6 s	0.0000	1.000

(b) Real-world workload; $K = 8 = 4 + 4$, $N = 344$, $Q = 4\,461$, cf. source [1]

Approach	S	F	$\frac{W}{V}$	solve time	$E(\tilde{L}) - \frac{1}{K}$	$E\left(\frac{1/K}{\tilde{L}}\right)$
$W(S)$	1	4 361	3.534	2.1 s	0.0435	0.769
$W(S)$	3	4 361	3.906	2.8 s	0.0308	0.830
$W(S)$	5	4 361	3.953	6.8 s	0.0264	0.851
$W(S)$	10	4 361	5.008	7.7 s	0.0141	0.921
$W(S)$	10	4 411	5.593	3.8 s	0.0048	0.971
$W(S)$	20	4 361	5.505	8.7 s	0.0007	0.995
$W(S)$	50	4 361	5.743	28.4 s	0.0001	0.999
$W(S)$	100	4 361	5.847	93.7 s	0.0001	0.999
$W(S)$	100	4 411	6.183	23.7 s	0.0000	1.000
$W^G(S)$	1	/	6.536	<3 s	0.0040	0.971
$W^G(S)$	3	/	6.792	<10 s	0.0013	0.991
$W^G(S)$	5	/	6.913	<16 s	0.0013	0.991
$W^G(S)$	10	/	7.040	<34 s	0.0000	1.000

which is by far better than the standard $S = 1$ solution (cf. $E^{(S)}(\tilde{L}) - 1/K = 0.0641$ for $F = 0$) that optimizes only against the expected load. Naturally, the higher robustness with $S = 10$ is achieved by using more data, i.e., a higher replication factor of 2.721 (in 562 s, $F = 0$) and 3.145 (in 20 s, $F = 47$), instead of 1.874 ($F = 0$) and 2.079 ($F = 47$) when using only one scenario ($S = 1$). Note, compared to achieving robustness via full replication (with $W/V = K = 8$), this is a remarkable result.

Further, we evaluated the robustness of the merge approach, cf. $W^G(S)$, for the same ($\tilde{S}$) unseen workloads. For $S = 5$, we obtained the average worst-case workload limit $E^{(G)}(\tilde{L}) - 1/K = 0.0041$ and replication factor $W^G(S)/V = 4.162$. Compared to that, our $S = 10$ solution with $F = 47$ fixed queries obtains a *better robustness* (cf. $E^{(S)}(\tilde{L}) - 1/K = 0.0037$) and requires *clearly less memory* $W^G(S)/V = 3.145$. The better combinations of memory (W/V) and robustness can be observed over the full range of S. Figure 2a visualizes this result: For selected S, the figure shows the expected throughput (average over $\tilde{S} = 100$ unseen workloads)



(a) Memory consumption vs. average relative throughput over 100 unseen scenarios.

(b) Relative throughput for all 100 unseen scenarios.

Figure 2: Performance of allocation approaches for $\tilde{S} = 100$ unseen workload scenarios based on TPC-DS; $K = 8$.

per memory consumption for full replication, the merge approach, and our partial clustering. For merged allocations with $S = 2$ and our partial clustering with $S = 10$ input scenarios, Figure 2b shows the expected throughput $(1/K)/\tilde{L}$ for all $\tilde{S} = 100$ individual unseen scenarios.

Naturally, the specific results depend on underlying random numbers. However, using repeated simulations, we verified that the overall properties remain.

4.2.2 Real-World Accounting Workload. The results for the real-world workload (see Table 3b) show that our solution also remains applicable for larger workloads. The number of fixed queries F can be chosen such that for up to 20 scenarios, the runtime is below 10 s. A good trade-off (depending on the targets in practice) between robustness, runtime, and memory is achieved for 5 to 20 in-sample scenarios, cf. S.

Compared to the merge approach, cf. $W^G(S)$, we again obtain that our approach clearly outperforms their combinations of memory and throughput robustness against uncertain workloads. Moreover, our approach provides the flexibility to *tune* results by adjusting S and F according to a decision-maker's preferences.

Compared to TPC-DS, we find that, for all S, the optimality gap $E^{(S)}(\tilde{L}) - 1/K$ is on average lower, while replication factors are higher. This difference indicates that, in such cases, a *smaller* number of scenarios might be necessary to obtain a certain robustness. It can be explained by the fact that the workload is distributed over a higher number of queries Q, making the impact of single query frequencies, on average, less important.

Remark 2 *We find that optimizing the memory for an expected workload only is not robust against unseen workloads. Less memory-efficient approaches as [12] and particularly its merge extension are (indirectly) more robust against out-of-sample workloads as they systematically allocate more data. However, our approach to directly minimize required memory for multiple seen workloads, yields a better robustness using the same or even less data. The memory-efficiency of our LP-based approach allows including more scenarios within a certain memory budget than the merge approach [12]. Being able to deal with more representative workload scenarios, in turn, allows to better deal with altered unseen ones.*

5 RELATED WORK

Database replication is a means to improve availability and increase processing capabilities, and is supported in many systems, e.g., in HANA [10], Postgres-R [8], and as replication middleware [3]. Thereby, most systems implement full replication.

To calculate partial allocations, Rabl and Jacobsen propose a greedy algorithm (Section 2.2.2). For the same problem, we propose an LP-based decomposition approach [5] (Section 2.2.3). In both papers, the authors only consider a comparably small benchmark workload (TPC-H) and do neither (i) derive results for large-size workloads, (ii) study techniques to reduce the computation time for allocations, nor (iii) evaluate robustness against uncertain workloads. In [6], we visualize calculated allocations and investigate the intermediate steps taken by different algorithms. In [7], we visualize the load balancing behavior of allocations.

Özsu and Valduriez present a general overview of allocation problems in the field of distributed database systems [11]. They point out that allocation problems differ in constraints and optimization goals, such as performance, costs, and dependability.

Archer et al. [2] address an allocation problem, which is similar to ours. They evenly load-balance queries for web search containing multiple terms, which correspond to the fragments in our model. They cluster queries with a distributed balanced graph partitioning tool.

6 CONCLUSIONS

To overcome the limitations of existing allocation approaches [5, 12], we proposed a novel heuristic that combines different concepts. First, to achieve *memory-efficiency*, instead of rule-based heuristics, we leverage the power of LP techniques. Second, for *short calculation times*, we exploit that (real-world) workloads are skewed and reduce the problem complexity by a partial clustering approach that assigns a majority of queries with comparably small workload shares to a fixed node. Third, to add *robustness*, we force the allocation to be prepared against multiple diversified workloads. Moreover, we are able to balance the three target dimensions of the problem *flexibly*. Using TPC-DS and a large real-world workload, we verified the applicability and effectiveness of our approach, which clearly outperformed existing approaches regarding the mix of the three key dimensions.

REFERENCES

[1] [n.d.]. GitHub repository containing the workload data and source code. https://hyrise.github.io/replication/.
[2] Aaron Archer et al. 2019. Cache-aware load balancing of data center applications. *PVLDB* 12, 6 (2019), 709–723.
[3] Emmanuel Cecchet et al. 2008. Middleware-based database replication: the gaps between theory and practice. In *SIGMOD*. 739–752.
[4] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. 1996. The Dangers of Replication and a Solution. In *SIGMOD*. 173–182.
[5] Stefan Halfpap et al. 2019. Workload-Driven Fragment Allocation for Partially Replicated Databases Using Linear Programming. In *ICDE*. 1746–1749.
[6] Stefan Halfpap and Rainer Schlosser. 2019. A Comparison of Allocation Algorithms for Partially Replicated Databases. In *ICDE*. 2008–2011.
[7] Stefan Halfpap and Rainer Schlosser. 2020. Exploration of Dynamic Query-Based Load Balancing for Partially Replicated Database Systems with Node Failures. In *CIKM*. 3409–3412.
[8] Bettina Kemme et al. 2000. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *VLDB*. 134–143.
[9] Jens Krüger et al. 2011. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB* 5, 1 (2011), 61–72.
[10] Juchang Lee et al. 2017. Parallel Replication across Formats in SAP HANA for Scaling Out Mixed OLTP/OLAP Workloads. *PVLDB* 10, 12 (2017), 1598–1609.
[11] M. Tamer Özsu and Patrick Valduriez. 2011. *Principles of Distributed Database Systems, Third Edition*. Springer.
[12] Tilmann Rabl and Hans-Arno Jacobsen. 2017. Query Centric Partitioning and Allocation for Partially Replicated Database Systems. In *SIGMOD*. 315–330.