Memory-Efficient Database Fragment Allocation for Robust Load Balancing when Nodes Fail

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

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

***Abstract*—Load balancing queries that access the same data fragments to the same node improves caching for a memory-efficient scale-out. However, to suitably allocate fragments to multiple nodes is a highly challenging problem, particularly when nodes might fail. The problem is to find a good balance between memory efficiency and allocating enough fragments to nodes to obtain robustness through load balancing flexibility. Existing allocation approaches are either not memory-efficient or result in load imbalances, both degrading cost/performance. In this paper, we present an optimal approach and a scalable heuristic, based on three mutually supportive linear programming models, to calculate memory-efficient fragment allocations that guarantee to distribute the workload evenly - even in the case of node failures. We demonstrate the applicability and the effectiveness of our three-step approach using numerical as well as end-to-end evaluations for TPC-H and TPC-DS workloads. We find that our robust solutions clearly outperform state-of-the-art heuristics by achieving a *better* workload distribution with even *less* memory.**

I. Introduction

Scalability and robustness against node failures are indispensable for database systems running in the cloud. Both can be achieved with query load balancing and data replication. Using a naive load balancing approach, queries are distributed independently of the accessed data. This approach has several drawbacks. All nodes have to store or potentially load all data. Further, all nodes must apply all data modifications caused by inserts, deletes, or updates. Finally, queries are unlikely to profit from caching effects, because similar queries are not guaranteed to be assigned to the same replica.

Query-driven workload distribution tackles this problem by load balancing queries to nodes of a cluster based on the accessed data. Such an approach is beneficial to optimize *cost/performance* [1], e.g., (i) in caching architectures, e.g., data marts [2] or mid-tier caches [3], (ii) for operator placement in distributed database systems, and (iii) for partially replicated database systems [4], [5]. In general, data can be allocated, i.e., stored or cached, at nodes such that the load can be evenly balanced, which is crucial for scalability, and data caching is optimized. However, in the presence of failures, in which the load of the failed node has to be distributed among the remaining nodes, memory-efficient data allocations may result in load imbalances, increased cache misses, or required reallocations. It is challenging to calculate memory-efficient allocations that guarantee an even workload distribution in failure cases, because we must consider potential failures of all nodes, at which different subsets of fragments are allocated and which are, thus, optimized for different subsets of queries.

This paper presents a general approach to calculate memory-efficient data allocations that guarantee to distribute the workload evenly - even in cases of node failures - using linear programming (LP). Although the developed allocation concepts are general, we focus on partial replication as one specific and thoroughly evaluated use case. The use of LP enables transferring our approach to versatile allocation problems by adapting the optimization goals and constraints [6].

Partial replication [4], [5] is a cost- and cache-efficient implementation of primary replication, which is a common scale-out option for single-node database systems. All major relational database systems, e.g., Oracle, IBM DB2, Microsoft SQL Server, SAP HANA, PostgreSQL, and MySQL, support replication. Partial replication lowers the memory consumption and improves caching, but reduces the robustness of a cluster, e.g., the accessibility of data fragments and executability of queries in the case of potential node failures.

Our contributions are: We present an optimal model and a scalable heuristic to calculate memory-efficient and robust fragment allocations that allow to evenly balance workloads – particularly in the case of potential node failures. Our heuristic (see Figure 2) exploits the optimal LP model for decomposed subproblems and uses minimal data enhancements to guarantee a perfect load balance. We verify the effectiveness of our models using theoretical and end-to-end evaluations for TPC-H and TPC-DS. We show that our approach finds allocations that outperform state-of-the-art approaches, cf. [5], [7], in both memory consumption and worst-case query throughput in theory (see Table I and II) and practice (see Figure 3).

II. Robust Fragment Allocation Problem

A. Problem Description

We study a system with K nodes, where one of them might fail. We assume a partitioned database with N fragments.

Input: The size of a fragment i is a_i, $i = 1, ..., N$. We assume K nodes, where data can be replicated. We assume a set of Q (classes of) queries j, characterized by the accessed fragments $q_j \subseteq \{1, ..., N\}$, $j = 1, ..., Q$. Queries j occur with frequency f_j, $j = 1, ..., Q$. Query costs c_j, $j = 1, ..., Q$, are independent of the executing node k. We use the total workload costs denoted by $C := \sum_{j=1,...,Q} f_j \cdot c_j$.

*Both authors contributed equally to this research.

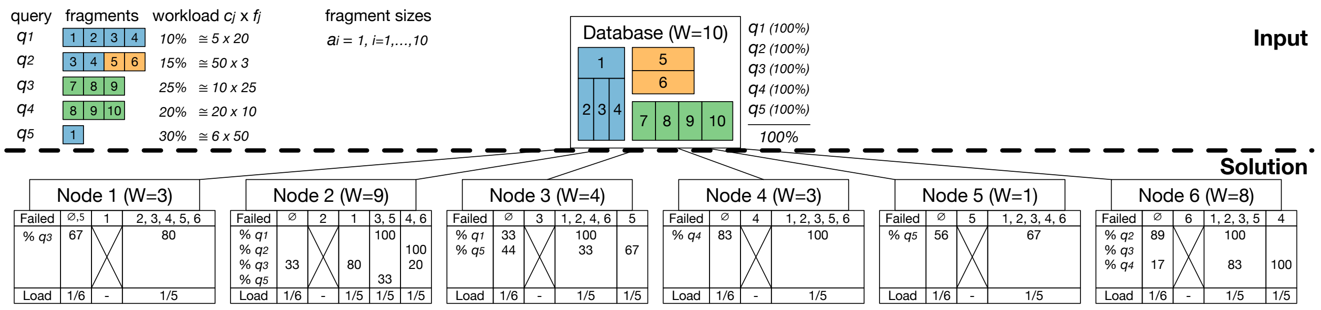


Fig. 1. Robust fragment allocation: compensating potential node failures by enabling different workload distributions. Illustration for $K = 6$ nodes, a regular workload limit $L^* = 1/6$ and a guaranteed worst-case workload limit (denoted by $L_{\max}^{(-)*}$) of 1/5. The total replication factor (denoted by W/V) is 2.8.

Controls: We use the following decision variables to decide (i) at which node to allocate which fragments and (ii) which query is executed on which node to which extent. $x_{i,k} \in \{0,1\}$, $i = 1, ..., N$, $k = 1, ..., K$, is allowed to be zero or one, indicating whether fragment i is allocated to node k (1) or not (0). $y_{j,k} \in \{0,1\}$, $j = 1, ..., Q$, $k = 1, ..., K$, is allowed to be zero or one, indicating whether query j can run on node k (1) or not (0). $z_{j,k} \in [0,1]$, $j = 1, ..., Q$, $k = 1, ..., K$, is indicating the workload share of query j executed on node k. Further, by W/V, we denote the *replication factor*, where the total amount of allocated data $W := \sum_{i=1,..,N,k=1,...,K} x_{i,k} \cdot a_i$ is normalized by the amount of relevant data $V := \sum_{i \in \bigcup_{j=1,...,Q: f_j > 0} \{q_j\}} a_i$.

Constraints: We have the following constraints: A query j can only be executed on node k if all relevant fragments are stored at node k. If all K nodes work, each node's load has to be $1/K$. If a node fails, it has to be possible to evenly balance the workload between all active nodes ($K - 1$).

Objective: We seek to minimize the total amount of allocated data for a feasible workload distribution. Data has to be placed on multiple nodes such that all queries are still executable in failure cases without overloading single nodes.

For an example with N=10 fragments and Q=5 queries, Figure 1 illustrates the solution for $K = 6$ nodes (lower part). The table below each node $k, k = 1, ..., 6$, specifies the query shares $z_{j,k}$ without a node failure (column ∅) and the query shares for each single node failure (columns 1 - 6).

B. Robust State-of-the-Art Heuristics

(i) Greedy Approach [5]. For allocations that do not base on LP, Rabl and Jacobson propose a *greedy* approach to complement a risk-neutral solution to a robust one: Queries are sorted by the size of the fragments they access in descending order. If a query is already executable by multiple nodes, nothing has to be done. Otherwise, the query is assigned to the node with the largest fragment overlap of already assigned queries, considering only the nodes that cannot already execute the query. The thereby added load of redundant query assignments in potential failure cases is not taken into account.

The load balancing among nodes may be *highly skewed* in failure cases. To an extreme, a single node must take over the entire workload of the failed node and cannot pass anything of its regular workload to other nodes.

(ii) Chaining Approach [7]. Allocations that *guarantee* an *even* workload distribution in failure cases can be constructed by applying the *chained declustering* strategy to a risk-neutral solution: Nodes are chained, forming a ring. The successor of each node is its backup. In addition to the fragments of the basic allocation, each (backup) node gets assigned all its predecessor's fragments. As a result, the backup can take over the complete assigned regular load of its predecessor and pass an arbitrary share of its regular workload to its successor.

(iii) Adding Full Replicas. To address cases with up to F node failures, one can use a basic solution for $K - F$ nodes and *add* F *full* replicas. If the F full replicas fail, the $K - F$ solution balances the load evenly by definition. Otherwise, remaining full replicas can take the workload of any failed partial node. However, full replicas can be memory-expensive.

III. Optimal Robust Solution

To solve the problem described in Section II-A, K potential failure cases and the non-failure case must be considered.

By L, we denote the highest workload share of all nodes in the regular case without a failure. Note, the limit L is determined by the allocation ($x_{i,k}$) and the assigned workload shares ($z_{j,k}$), see Section II-A. By $L^{(-)}$, we denote the highest (worst-case) workload share over all nodes that can occur in case of potential node failures. To optimize the limit $L^{(-)}$, we introduce the additional variables $\tilde{z}_{j,k^{(-)},k^{(+)}} \in [0,1]$, which describe the adjusted workload share of query j on a remaining node $k^{(+)} \in \{1, ..., K\} \backslash \{k^{(-)}\}$ in case node $k^{(-)} = 1, ..., K$ fails. In the tables of Figure 1, $\tilde{z}$ refers to the columns 1-6.

To obtain solutions with workload limits L (regular case) and $L^{(-)}$ (failure case) that are *as small as possible*, in our LP model, we use a common penalty approach, where both limits are penalized via one penalty factor (denoted by $\alpha > 0$). To emphasize the failure scenarios, the penalty on $L^{(-)}$ is chosen much larger (e.g., $\times 100$) than the penalty for L. Our LP for optimal robust solutions with single node failures reads as:

$$\underset{\substack{x_{i,k}, y_{j,k} \in \{0,1\}, z_{j,k}, \tilde{z}_{j,k^{(-)},k^{(+)}} \in [0,1], L, L^{(-)} \geq 0, \\ i=1,...,N, j=1,...,Q, k, k^{(-)}, k^{(+)}=1,...,K, k^{(+)} \neq k^{(-)}}}{minimize}$$

$$1/V \cdot \sum_{i=1,...,N,k=1,...,K} a_i \cdot x_{i,k} + \alpha \cdot L^{(-)} + \alpha/100 \cdot L \quad (1)$$

subject to constraints for the *regular* scenario:

$$y_{j,k} \cdot |q_j| \leq \sum\nolimits_{i \in q_j} x_{i,k}, \quad j = 1, ..., Q, k = 1, ..., K \tag{2}$$

$$z_{j,k} \leq y_{j,k}, \quad j = 1, ..., Q, k = 1, ..., K \tag{3}$$

$$\sum\nolimits_{j=1,...,Q} f_j \cdot c_j / C \cdot z_{j,k} \leq L, \quad k = 1, ..., K \tag{4}$$

$$\sum\nolimits_{k=1,...,K} z_{j,k} = 1, \quad j = 1, ..., Q \tag{5}$$

and constraints for the *failure* scenarios:

$$\tilde{z}_{j,k^{(-)},k^{(+)}} \leq y_{j,k^{(+)}}, \quad \begin{array}{l} j = 1, ..., Q, k^{(-)} = 1, ..., K \\ k^{(+)} \in \{1, ..., K\} \backslash \{k^{(-)}\} \end{array} \tag{6}$$

$$\sum_{j=1,...,Q} \frac{f_j \cdot c_j}{C} \tilde{z}_{j,k^{(-)},k^{(+)}} \leq L^{(-)}, \quad \begin{array}{l} 1 \leq k^{(-)}, k^{(+)} \leq K \\ k^{(+)} \neq k^{(-)} \end{array} \tag{7}$$

$$\sum_{k^{(+)}=\{1,...,K\} \backslash \{k^{(-)}\}} \tilde{z}_{j,k^{(-)},k^{(+)}} = 1, \quad \begin{array}{l} j = 1, ..., Q \\ 1 \leq k^{(-)} \leq K \end{array} \tag{8}$$

Objective (1) minimizes the replication factor W/V and contains a penalty term for the largest workload shares L and $L^{(-)}$. (2) guarantees that a query j can only be executed on node k if all relevant fragments are available, see Section II. The cardinality term $|q_j|$ expresses the number of fragments used in query j. (3) ensures that a query j can only have a positive workload share on node k if it can be executed on node k. If $y_{j,k} = 0$ then $z_{j,k} = 0$ follows; if $y_{j,k} = 1$ the shares $z_{j,k}$ are not restricted. Note, (3) couples the binary variables y and the continuous variables z in a linear way. (4) guarantees that all nodes k do not exceed the workload limit L. (5) ensures that a query's workload shares on nodes k sum up to one.

Constraints (6)-(8) ensure admissible allocations in case a node $k^{(-)} = 1, ..., K$ is not working. The additional variables $\tilde{z}_{j,k^{(-)},k^{(+)}} \in [0, 1]$ express the optimal workload share of query j on the remaining nodes $k^{(+)} = \{1, ..., K\} \backslash \{k^{(-)}\}$ if node $k^{(-)}$ does not work. If the penalty α is sufficiently large, the solution guarantees the smallest possible emergency workload limit $L^{(-)*} = 1/(K - 1)$, $K > 1$. As all scenarios are mutually coupled, they cannot be optimized independently.

We numerically evaluate the model for the TPC-H and TPC-DS setup described in Section V-A. Table I summarizes the results of our optimal solution (cf. W^{R*}) and compares the memory consumption and worst-case workload shares against the greedy robust heuristic (cf. W^{GR}) and the chaining approach (cf. W^{CR}) based on risk-neutral allocations of [5]. To calculate the worst-case workload limits $L_{max}^{GR(-)}$, we can use the LP (1) - (8) with a fixed fragment allocation $\vec{x} := \vec{x}^{GR}$. We used the Gurobi solver (version 9.0.0) (single-threaded on a laptop) with $\alpha := 1\,000$, cf. (1).

The results of Table I show that our solution outperforms the greedy robust heuristic [5] in *both* required memory (up to 24.9% less) and worst-case workload share (up to 42.5% lower). Note, $W^{GR} < W^{R*}$ is only possible, because $L_{max}^{GR(-)}$ is worse than the optimal limit $L^{(-)*}$, cf. K=4, TPC-H. Compared to the chained heuristic [7], our solution lowers the memory consumption more substantially (up to 41.8% less) while providing the same worst-case limits. As the complexity quickly increases, the LP is only applicable to small problems.

TABLE I
OUR OPTIMAL ROBUST SOLUTION (W^{R*}) VS. THE GREEDY HEURISTIC [5] (W^{GR}) & THE CHAINED HEURISTIC [7] (W^{CR}): DATA REPLICATION FACTOR W^{R*}/V AND WORST-CASE WORKLOAD LIMIT $L^{(-)*}$ FOR DIFFERENT K; #SOLVED WITH RELAXED OPTIMALITY GAP 10^{-3}.

K	$\frac{W^{R*}}{V}$	$L^{(-)*}$	time$_{W^{R*}}$	$\frac{W^{R*}}{W^{GR}}$	$\frac{L^{(-)*}}{L_{max}^{GR(-)}}$	$\frac{W^{R*}}{W^{CR}}$
3	2.371	0.500	0.2 s	- 5.3%	+ 0.0%	-14.8%
4	2.651	0.333	1.5 s	+ 3.7%	-32.4%	-15.0%
5	2.886	0.250	5.7 s	- 2.3%	-16.2%	-21.7%
6	3.153	0.200	29.1 s	- 5.5%	-11.4%	-23.3%
7	3.411	0.167	40.3 s	- 4.9%	-32.9%	-26.1%
8	3.708	0.143	3 507 s	- 0.9%	-42.5%	-21.9%
9	3.912	0.125	8 551 s	- 2.5%	-40.4%	-25.1%

(a) TPC-H

K	$\frac{W^{R*}}{V}$	$L^{(-)*}$	time$_{W^{R*}}$	$\frac{W^{R*}}{W^{GR}}$	$\frac{L^{(-)*}}{L_{max}^{GR(-)}}$	$\frac{W^{R*}}{W^{CR}}$
3	2.126	0.500	16.9 s	- 0.2%	-20.9%	-22.3%
4	2.204	0.333	94.0 s	-20.2%	- 8.6%	-36.2%
5	2.296	0.250	573 s	-17.8%	- 4.0%	-37.7%
6	2.390	0.200	21 248 s	-21.8%	-24.5%	-41.6%
7#	2.454#	0.167	3 958 s	-24.9%	- 2.2%	-41.8%

(b) TPC-DS

Remark 1 *Existing robust allocation approaches have limitations. We find that with allocations of the greedy robust heuristic [5], the load balance can be uneven if nodes fail. In contrast, the chained heuristic [7] is memory-expensive. As the optimal solution does not scale, a heuristic is needed that combines both reliable robustness and memory efficiency.*

IV. Heuristic Three-Step Solution

We propose a three-step heuristic to solve problem (1)-(8).

A. Step 1: Initial Recursive Chunking

In Step 1, we split the workload iteratively using a risk-neutral LP approach described in [4], forming a tree of chunks with similar queries, which access the same fragments. We specify the assigned workload of a chunk by the (fixed) parameters $\bar{x}_i$, $\bar{y}_j \in \{0, 1\}$, $\bar{z}_j \in [0, 1]$, $i = 1, ..., N$, $j = 1, ..., Q$. We only consider (i) involved fragments i, where $\bar{x}_i = 1$ and (ii) queries j, where $\bar{z}_j > 0$. The node's workload share is denoted by $\bar{w} := \sum_{j=1,...,Q: \bar{y}_j = 1} f_j \cdot c_j / C \cdot \bar{z}_j$. We split a chunk's workload share $\bar{w}$ into B subnodes, where a subnode b represents n_b leaf nodes with a workload share $w_b := n_b/K$, $\sum_{b=1,..,B} w_b = \bar{w}$, using the LP given in [4], cf. (1)-(5). Assume the optimal solution of a split is x^*, y^*, and z^*. Then, for each subnode b, we use the remaining fragments and workload share as input on the next level and apply the same LP again. The smaller B is chosen, the faster is the computation (on each level) but the data redundancy of the heuristic might raise compared to optimal allocations.

B. Step 2: Robustness on the Final Level

Assume, on the final decomposition level of Step 1, we end up with B chunks, where the data allocation for each chunk b, $b = 1, ..., B$, with weights $\bar{w}_b = n_b/K$ is characterized by (fixed) values $\bar{x}_{i,b}$, $\bar{y}_{j,b}$, $\bar{z}_{j,b}$. In Step 2, we proceed as follows:

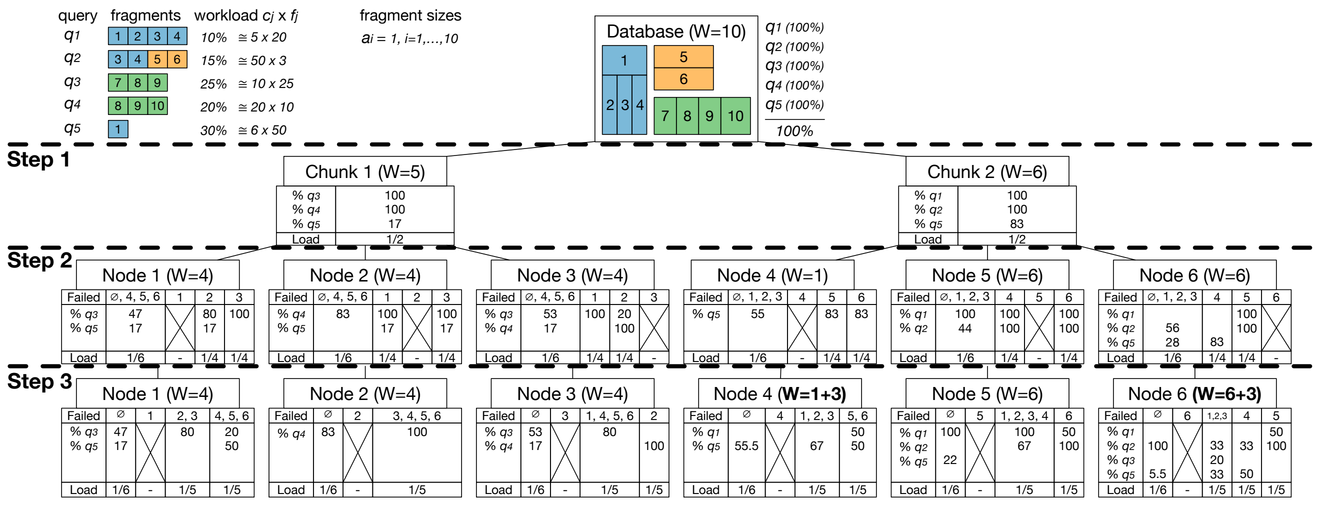


Fig. 2. Robust three-step approach: splitting the workload into chunks in Step 1; compensating potential node failures within chunks in Step 2; optimal fragment enhancement in Step 3; illustration for $K = 6$ nodes, $B = 2$ chunks, a regular workload limit $L = 1/6$ and a guaranteed worst-case workload limit of $L^{(-)}_{\max} = 1/5$. After Step 2, we have $W^R/V = 2.5$; the total replication factor is $W^R/V = 3.1$ (cp. $W^{R*}/V = 2.8$ is optimal, see Figure 1).

For each chunk b, $b = 1, ..., B$, we solve the problem (1)-(8) for the corresponding (smaller) inputs. Figure 2 shows an example of Step 2, which splits B=2 chunks in overall K=6 nodes (three nodes per chunk). The table below each node k specifies the regular workload distribution $z_{j,k}$ (cf. column $\varnothing$) and six individual emergency load distributions $\tilde{z}_{j,k^{(-)},k^{(+)}}$ (cf. columns 1-6) for each potential node failure $k^{(-)} = 1, ..., 6$.

After Step 2, the allocation is such that the emergency load distributions are only affected for failures of nodes from the same chunk (cf. Chunk 1). In such cases (e.g., $k^{(-)} = 2$), the workload distribution within the affected Chunk 1 is reorganized by evenly balancing the load between the remaining two nodes Node 1 and 3 (load 1/4). The other chunk's nodes retain their regular workload distribution (load 1/6).

C. Step 3: Optimal Fragment Enhancements

The final Step 3 seeks to enrich the fragment allocation derived in Step 2 to obtain a perfect load balancing over all nodes in the case of any node failure. The goal is to use the minimal amount of additional data to guarantee the best possible workload limit ($L^{(-)*} = 1/(K-1)$). Let $x^{(f)}_{i,k} \in \{0, 1\}$ denote the allocation of Step 2, i.e., whether fragment $i = 1, ..., N$ is assigned to node $k = 1, ..., K$. We consider this allocation as *fixed*. In the LP (1)-(8), we use the *new* decision variables $x^{(e)}_{i,k} \in \{0, 1\}$ to decide whether to also add fragment i to node k. This variable $x^{(e)}_{i,k}$ is used in the objective (1) instead of $x_{i,k}$. Further, we add the constraints

$$x^{(f)}_{i,k} + x^{(e)}_{i,k} = x_{i,k}, \quad i = 1, ..., N, k = 1, ..., K \qquad (9)$$

to define the actual fragments as the compound of fixed and new fragment. Compared to (1)-(8), the modified LP for Step 3 is of much lower complexity, as $x^{(f)}_{i,k}$ are given parameters, i.e., those fragments cannot be removed. Further, the freedom of the enhancement variables $x^{(e)}_{i,k} = 0$ is limited, as (9) implies $x^{(e)}_{i,k} = 0$ for all $x^{(f)}_{i,k} = 1$. The other variables x, y, z, and $\tilde{z}$ are of auxiliary character, as they are governed by $x^{(e)}_{i,k}$.

In Step 3, comparably little data has to be assigned in total, because the allocations $x^{(f)}_{i,k}$ derived in Step 2 have the following beneficial properties: Assume an arbitrary solution of Step 2 with two chunks, say chunk C_1 and C_2. To obtain a perfect load balancing if a node of C_1 fails, the nodes of C_2 have to take additional load of C_1. However, as C_2 only has to be able to take *some arbitrary* load of C_1, it is sufficient to look for *arbitrary* nodes of C_2 that can be efficiently completed in this regard via additional fragments. Due to this flexibility, typically only *little* additional data is necessary. Finally, if one node of C_1 fails, the remaining nodes of C_1 as well as the nodes of C_2 can easily/flexibly compensate the additional load, as within Step 2 all chunks are exactly optimized for such scenarios. Hence, for multiple chunks, it is sufficient when each chunk can take and pass enough load to *one* other chunk, and all are connected.

In Figure 2, Node 4 is enhanced with three fragments (2-4) for q_1, and three fragments (7-9) for q_3 are added to Node 6. After Step 3, we obtain that, whatever node fails, a perfect workload distribution can always be achieved (load 1/5). The final replication factor $W^R/V = 3.1$ is close to the optimal solution $W^{R*}/V = 2.8$ (see Figure 1).

Note, base allocations $x^{(f)}_{i,k}$ are a prerequisite for the applicability of Step 3, as without a suitable (already robust) backbone solution, the LP (1) - (8) might be too complex.

V. Evaluation

After a description of our end-to-end evaluation setup, we compare our approaches against the results of [5] and [7].

A. Setup and Model Input

We set up a replicated PostgreSQL cluster with 16 nodes for running TPC-H and TPC-DS queries with scale factor 1. In the following, we describe how we obtained the model inputs. For TPC-H (Q = 22) and TPC-DS (Q = 99), we modeled query costs c_j as average processing time of query j with random template parameters, $j = 1, ..., Q$. We deployed single-column

TABLE II
PERFORMANCE COMPARISON OF ROBUST ALLOCATIONS: OUR THREE-STEP HEURISTIC (W^R) VS. THE ROBUST HEURISTIC BY [5] (W^{GR}) AND CHAINING [7] (W^{CR}). $^{(2)}$ RESULT AFTER STEP 2.

K	chunks	$\frac{W^R}{V}$	$L_{max}^{(-)}$	time$_{W^R}$	$\frac{W^R}{W^{GR}}$	$\frac{L_{max}^{(-)}}{L_{max}^{GR(-)}}$	$\frac{W^R}{W^{CR}}$
8	4+4	3.947	0.143	5.5 s	+5.5%	-42.5%	-16.9%
9	3+3+3	4.305	0.125	2.5 s	+7.2%	-40.4%	-17.5%
$10^{(2)}$	5+5	4.481	0.125	14.0 s	-0.4%	-12.9%	-19.5%
10	5+5	4.524	0.111	14.8 s	+0.5%	-22.6%	-18.8%
11	6+5	4.611	0.100	42.7 s	-5.5%	-30.3%	-19.6%
12	6+6	4.982	0.091	27.4 s	-3.1%	-36.7%	-16.9%
13	7+6	5.430	0.083	16.2 s	+3.8%	-41.9%	-13.5%
14	5+5+4	5.396	0.077	12.2 s	-7.2%	-29.8%	-21.7%
15	5+5+5	5.790	0.071	6.9 s	-1.0%	-28.1%	-16.4%
$16^{(2)}$	8+8	6.027	0.071	139 s	-3.7%	-28.5%	-20.0%
16	8+8	6.105	0.067	151 s	-2.5%	-32.8%	-19.0%

(a) TPC-H

K	chunks	$\frac{W^R}{V}$	$L_{max}^{(-)}$	time$_{W^R}$	$\frac{W^R}{W^{GR}}$	$\frac{L_{max}^{(-)}}{L_{max}^{GR(-)}}$	$\frac{W^R}{W^{CR}}$
5	3+2	2.443	0.250	25.8 s	-12.5%	- 4.0%	-33.8%
6	3+3	2.550	0.200	24.3 s	-16.6%	-24.5%	-37.7%
7	4+3	2.624	0.167	92.8 s	-19.7%	- 2.2%	-37.7%
8	4+4	2.683	0.143	66.6 s	-18.7%	-5.7%	-36.9%
9	3+3+3	3.017	0.125	16.3 s	-10.6%	-11.9%	-30.2%
10	4+3+3	3.102	0.111	14.4 s	-11.7%	- 6.3%	-32.7%
11	4+4+3	3.175	0.100	58.0 s	- 8.9%	-10.2%	-32.2%
12	4+4+4	3.274	0.091	86.2 s	-11.8%	-31.5%	-35.0%
13	4+3+3+3	3.352	0.083	93.3 s	-11.9%	-27.0%	-36.6%
14	4+4+3+3	3.636	0.077	176 s	- 8.4%	- 7.5%	-33.2%
$15^{(2)}$	4+4+4+3	3.507	0.100	111 s	-15.6%	+19.7%	-38.2%
$15^{(2)}$	5+5+5	3.430	0.083	895 s	-17.5%	- 0.3%	-39.6%
15	4+4+4+3	3.682	0.071	124 s	-11.4%	-14.5%	-35.2%
16	4+4+4+4	4.044	0.067	209 s	- 6.9%	-19.8%	-30.8%

(b) TPC-DS

indices on all primary key columns. As processing TPC-H query 17 and 20 exceeded the set timeout of 120 s, we omitted them in our allocations. Likewise, we omitted the five most expensive TPC-DS queries, resulting in 94 remaining queries. We use vertical partitioning with each column as an individual fragment. Fragment/column sizes $a_i, i = 1, ..., N$, for TPC-H ($N = 61$) and TPC-DS ($N = 425$) are modeled by using the PostgreSQL function `pg_column_size()`. In case there is an index on an attribute, the associated fragment size is increased by the index size. All model inputs to reproduce the calculation of all allocations are available online [8].

B. Numerical Evaluation of Step 2 and Step 3

Table II compares the results of our robust heuristic (cf. W^R) against the greedy [5] (cf. W^{GR}) and chaining approach [7] (cf. W^{CR}). The results verify that also large problems can be addressed in a reasonable time.

For TPC-H (Table IIa), we find that our worst-case limits $L_{max}^{(-)}$ are *clearly better* (up to 42.5%) compared to the greedy approach, although our heuristic requires similar or even less memory. To illustrate the impact of Step 3, we also include results obtained after Step 2 (indicated by $^{(2)}$), cf. $K = 10, 16$. We observe that the amount of data enhancements to realize an optimal load balancing is small. After Step 3, for all K the limit $L_{max}^{(-)}$ coincides with the optimal lower bound $L^{(-)*} = 1/(K-1)$. In contrast, in some settings (e.g., TPC-H, $K = 8$, 9, or 13) the limit $L_{max}^{GR(-)}$ of [5] can be close to the worst case $2 \times L^*$ (see, e.g., TPC-H, $K = 8$ where $L_{max}^{GR(-)} = 0.248$ and $2 \times L^* = 2/K = 0.250$), which reflects the case in which one node has to (i) additionally take the entire workload of the failure node ($1/K$) and (ii) cannot pass some of its regular workload ($1/K$) to other nodes. Compared to the chaining approach, our three-step approach requires less memory (up to 21.7%) for all $K \geq 3$, while providing the same worst-case limits, i.e, $L_{max}^{(-)} = L_{max}^{CR(-)}$.

For TPC-DS (Table IIb), we observe that our heuristic constantly requires significantly *less memory* (up to 19.7%) than the greedy approach while still obtaining *better (optimal)* worst-case limits (up to 31.5%). Again, the data enhancements and additional runtime of Step 3 are small. Compared to optimal solutions (cf. Table Ib, $K = 5, 6, 7$) our heuristic's memory consumption is near-optimal and, most importantly, remains applicable to larger numbers K. Compared to the chaining approach with the same worst-case limits, our three-step approach requires 30-38% less memory for all $K \geq 5$.

To avoid long runtimes, we can use Step 2 with smaller chunks (see K=4+4+4+3 vs. K=5+5+5, TPC-DS), which can be derived significantly faster (111 s vs. 895 s) while the required memory is only slightly (2%) higher.

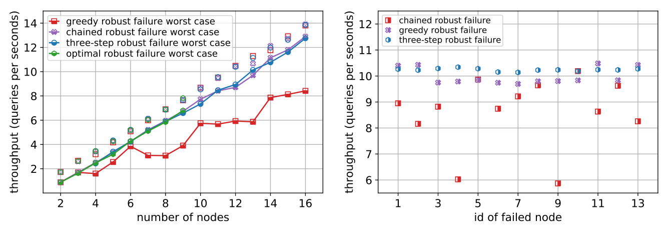
Recall, the greedy approach [5] does not optimize the limit $L_{max}^{GR(-)}$, and is, hence, to some extent *unforeseeable* regarding its quality (see, e.g., K=7 vs. K=12, TPC-DS). In contrast, the chaining approach [7] does not focus on memory efficiency. Chaining entire (and mostly unrelated) nodes wastes optimization potential compared to both adding robustness per chunk and more fine granular fragment enhancements (Step 3).

Remark 2 *The evaluation shows that our three-step heuristic clearly outperforms both robust approaches [5] and [7] when comparing (i) memory efficiency (up to 38% better) and (ii) worst-case workload limits (up to 42% better). Further, even large problems can be solved in a reasonable time.*

The quality of our results is based on the mutually supportive interplay of the three LP models, cf. Step 1 - 3: Step 1 reduces the complexity of the initial problem using a memory-efficient workload decomposition. Exploiting the LP (1) - (8), Step 2 effectively adds robustness within the final chunks of Step 1. Finally, based on Step 2's allocation, Step 3 guarantees a perfect load balance using optimal data enhancements.

C. End-To-End Evaluation

We evaluate the TPC-H throughput of allocations in a PostgreSQL cluster. The number of benchmark streams $S = 8 \cdot K$ (representing users) depends on the cluster size K. A central dispatcher maintains a query queue for each replica. For full replication, queries from stream s are added to the queue of node $k = 1 + (s-1) \mod K$. For partial replication, queries are added to a queue of a node which stores all relevant fragments to process the query (considering the costs of queued and currently processed queries). A fixed number of connections per replica is used to query the database, removing queries from the according queue. For K=16, there are $8 \cdot 16 = 128$ clients, resulting in 128 active queries at a time. We run an



(a) Regular and worst-case throughput of all scenarios for $K = 2, \ldots, 16$.

(b) Throughput of all single failure scenarios for $K = 13$.

Fig. 3. End-to-end TPC-H throughput of allocations in a PostgreSQL cluster.

experiment with each setting for 620 seconds, executing more than 8 000 TPC-H queries. We started measuring the query throughput after a 180 seconds warm-up phase.

For each number of nodes K and each allocation, we evaluate $K + 1$ scenarios: the case with no failure and K scenarios in which node k failed, i.e., node k is not used for query processing. Figure 3a shows the query throughput without a failure (regular) and the measured minimum (worst-case) performance of all failure scenarios. The end-to-end results correspond to the numerical results of Table Ia and IIa: (i) The optimal robust allocations provide the overall best results, having high throughput despite arbitrary single-node failures with the lowest memory consumption (see Table Ia). (ii) Using slightly more data, our three-step approach also allows calculating allocations for large cluster sizes with the same throughput properties as the optimal solution. (iii) Our allocations provide a clearly higher worst-case throughput for larger K (*up to +91%* for K=8) than solutions by [5], which have a similar memory consumption (see Table IIa). (iv) Recall, allocations of the chaining approach require significantly more memory (*up to +28%* for K=14) than our approach.

Figure 3b visualizes the throughput in all failure cases for the cluster size $K = 13$. The chaining and our robust approach provide high and stable throughput in all failure cases, whereas the throughput of Rabl and Jacobsen's allocations may drop significantly when a specific node (e.g., 4 or 9) fails.

Remark 3 *The conducted end-to-end evaluation (Figure 3) verifies that the theoretically obtained results for our fragment allocations, i.e., memory efficiency and optimal worst-case workload limits, also hold in deployed systems.*

VI. Related Work

Özsu and Valduriez give an overview of allocation problems in the context of distributed database systems [6]: Allocation problems differ in (i) optimization goals, e.g., performance, costs, and reliability, and (ii) constraints based on the system assumptions. Because problem formulations are often proven to be NP-hard, a lot of research tries to find good heuristic solutions. As optimization goals and constraints differ, heuristics are often tied to specific allocation problems.

Our optimization goal and the constraints are similar to the work of Rabl and Jacobsen [5]. We maximize throughput by balancing the load and minimize the cluster's overall memory consumption. Rabl and Jacobsen showed that partial replication does not only reduces the memory consumption for read-intensive workloads, but also scales better for write-intensive workloads, because replicas have to modify only stored fragments. In contrast to the robust extension in [5], our approach decomposes the problem, adds robustness, and enhances the solution, using linear programming for all steps.

Archer et al. address a similar (coupled data and query assignment) problem [9]. They evenly load balance queries for web search containing multiple terms, which correspond to the fragments in our model. In contrast to our model, data of assigned terms (fragments) can be loaded on demand. In contrast, the allocation problem tackled by Ghosh et al. [10] has no coupling, which reduces the complexity of the problem significantly. They replicate fragments according to the access rate and balance the number of fragments per node. Further, they focus on a dynamic setting [11], in which fragments and queries change over time. Allocation problems with uncertain workloads are heuristically addressed in [12].

VII. Conclusions

This paper investigated a problem to evenly balance a workload among nodes to maximize throughput while minimizing the cluster's overall memory consumption. Taking single node failures into account, data fragments have to be assigned to nodes such that a given workload can be evenly balanced in any scenario. Besides an LP-based optimal solution, which is only applicable to small problems, we presented a scalable three-step heuristic. We compared our robust approaches with the state-of-the-art techniques [5] and [7] for the TPC-H and TPC-DS workload. Using numerical and end-to-end evaluations, we showed that our three-step heuristic calculates close to optimal allocations and outperforms current techniques by achieving *better* combinations of worst-case throughput and required memory consumption, e.g., increasing the end-to-end throughput by up to 91% compared to approach [5], and using up to 38% less memory than approach [7].

References

[1] D. B. Lomet, "Cost/performance in modern data stores: how data caching systems succeed," in *DaMoN*, 2018, pp. 9:1–9:10.

[2] A. Bonifati *et al.*, "Designing data marts for data warehouses," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 4, pp. 452–483, 2001.

[3] Q. Luo *et al.*, "Middle-tier database caching for e-business," in *SIGMOD*, 2002, pp. 600–611.

[4] S. Halfpap and R. Schlosser, "Workload-driven fragment allocation for partially replicated databases using linear programming," in *ICDE*, 2019, pp. 1746–1749.

[5] T. Rabl and H. Jacobsen, "Query centric partitioning and allocation for partially replicated database systems," in *SIGMOD*, 2017, pp. 315–330.

[6] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[7] H. Hsiao *et al.*, "Chained declustering: A new availability strategy for multiprocessor database machines," in *ICDE*, 1990, pp. 456–465.

[8] "Code, data, and scripts for reproducibility of results," https://hyrise.github.io/replication/.

[9] A. Archer *et al.*, "Cache-aware load balancing of data center applications," *PVLDB*, vol. 12, no. 6, pp. 709–723, 2019.

[10] M. Ghosh *et al.*, "Popular is cheaper: curtailing memory costs in interactive analytics engines," in *EuroSys*, 2018, pp. 40:1–40:14.

[11] R. Marcus, O. Papaemmanouil, S. Semenova, and S. Garber, "Nashdb: An end-to-end economic method for elastic database fragmentation, replication, and provisioning," in *SIGMOD*, 2018, pp. 1253–1267.

[12] R. Schlosser *et al.*, "Robust and memory-efficient database fragment allocation for large and uncertain database workloads," in *EDBT*, 2021.