Hasso-Plattner-Institut für Digital Engineering
Enterprise Platform and Integration Concepts

Integer Linear Programming-Based Heuristics for Partially Replicated Database Clusters and Selecting Indexes

---

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
in der Wissenschaftsdisziplin *Praktische Informatik*

eingereicht an der
Digital Engineering Fakultät
der Universität Potsdam

von
Stefan Halfpap (geb. Klauck)

Betreuer:

Prof. Dr. h.c. mult. Hasso Plattner

Gutachter:

Prof. Dr. Hans-Arno Jacobsen
Prof. Dr. Kai-Uwe Sattler

Potsdam, 27. April 2023

Published online on the
Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup-63361
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-633615

Abstract

Column-oriented database systems can efficiently process transactional and analytical queries on a single node. However, increasing or peak analytical loads can quickly saturate single-node database systems. Then, a common scale-out option is using a database cluster with a single primary node for transaction processing and read-only replicas. Using (the naive) full replication, queries are distributed among nodes independently of the accessed data. This approach is relatively expensive because all nodes must store all data and apply all data modifications caused by inserts, deletes, or updates.

In contrast to full replication, partial replication is a more cost-efficient implementation: 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. Besides lower storage costs, partial replicas enable (i) better scaling because replicas must potentially synchronize only subsets of the data modifications and thus have more capacity for read-only queries and (ii) better elasticity because replicas have to load less data and can be set up faster. However, splitting the overall workload evenly among the replica nodes while optimizing the data allocation is a challenging assignment problem.

The calculation of optimized data allocations in a partially replicated database cluster can be modeled using integer linear programming (ILP). ILP is a common approach for solving assignment problems, also in the context of database systems. Because ILP is not scalable, existing approaches (also for calculating partial allocations) often fall back to simple (e.g., greedy) heuristics for larger problem instances. Simple heuristics may work well but can lose optimization potential.

In this thesis, we present optimal and ILP-based heuristic programming models for calculating data fragment allocations for partially replicated database clusters. Using ILP, we are flexible to extend our models to (i) consider data modifications and reallocations and (ii) increase the robustness of allocations to compensate for node failures and workload uncertainty. We evaluate our approaches for TPC-H, TPC-DS, and a real-world accounting workload and compare the results to state-of-the-art allocation approaches. Our evaluations show significant improvements for varied allocation's properties: Compared to existing approaches, we can, for example, (i) almost halve the amount of allocated data, (ii) improve the throughput in case of node failures and workload uncertainty while using even less memory, (iii) halve the costs of data modifications, and (iv) reallocate less than 90% of data when adding a node to the cluster. Importantly, we can calculate the corresponding ILP-based heuristic solutions within a few seconds. Finally, we demonstrate that the ideas of our ILP-based heuristics are also applicable to the index selection problem.

Zusammenfassung

Spaltenorientierte Datenbanksysteme können transaktionale und analytische Abfragen effizient auf einem einzigen Rechenknoten verarbeiten. Steigende Lasten oder Lastspitzen können Datenbanksysteme mit nur einem Rechenknoten jedoch schnell überlasten. Dann besteht eine gängige Skalierungsmöglichkeit darin, einen Datenbankcluster mit einem einzigen Rechenknoten für die Transaktionsverarbeitung und Replikatknoten für lesende Datenbankanfragen zu verwenden. Bei der (naiven) vollständigen Replikation werden Anfragen unabhängig von den Daten, auf die zugegriffen wird, auf die Knoten verteilt. Dieser Ansatz ist relativ teuer, da alle Knoten alle Daten speichern und alle Datenänderungen anwenden müssen, die durch das Einfügen, Löschen oder Aktualisieren von Datenbankeinträgen verursacht werden.

Im Gegensatz zur vollständigen Replikation ist die partielle Replikation eine kostengünstige Alternative: Anstatt alle Daten auf alle Replikationsknoten zu duplizieren, speichern partielle Replikate nur eine Teilmenge der Daten und können gleichzeitig einen großen Anteil der Anfragelast verarbeiten. Neben niedrigeren Speicherkosten ermöglichen partielle Replikate (i) eine bessere Skalierung, da Replikate potenziell nur Teilmengen der Datenänderungen synchronisieren müssen und somit mehr Kapazität für lesende Anfragen haben, und (ii) eine bessere Elastizität, da Replikate weniger Daten laden müssen und daher schneller eingesetzt werden können. Die gleichmäßige Lastbalancierung auf die Replikatknoten bei gleichzeitiger Optimierung der Datenzuweisung ist jedoch ein schwieriges Zuordnungsproblem.

Die Berechnung einer optimierten Datenverteilung in einem Datenbankcluster mit partiellen Replikaten kann mithilfe der ganzzahligen linearen Optimierung (engl. integer linear programming, ILP) durchgeführt werden. ILP ist ein gängiger Ansatz zur Lösung von Zuordnungsproblemen, auch im Kontext von Datenbanksystemen. Da ILP nicht skalierbar ist, greifen bestehende Ansätze (auch zur Berechnung von partiellen Replikationen) für größere Probleminstanzen oft auf einfache Heuristiken (z.B. Greedy-Algorithmen) zurück. Einfache Heuristiken können gut funktionieren, aber auch Optimierungspotenzial einbüßen.

In dieser Arbeit stellen wir optimale und ILP-basierte heuristische Ansätze zur Berechnung von Datenzuweisungen für partiell-replizierte Datenbankcluster vor. Mithilfe von ILP können wir unsere Ansätze flexibel erweitern, um (i) Datenänderungen und -umverteilungen zu berücksichtigen und (ii) die Robustheit von Zuweisungen zu erhöhen, um Knotenausfälle und Unsicherheiten bezüglich der Anfragelast zu kompensieren. Wir evaluieren unsere Ansätze für TPC-H, TPC-DS und eine reale Buchhaltungsanfragelast und vergleichen die Ergebnisse mit herkömmlichen Verteilungsansätzen. Unsere Auswertungen zeigen signifikante Verbesserungen für verschiedene Eigenschaften der berechneten Datenzuordnungen: Im Vergleich zu beste-

henden Ansätzen können wir beispielsweise (i) die Menge der gespeicherten Daten in Cluster fast halbieren, (ii) den Anfragedurchsatz bei Knotenausfällen und unsicherer Anfragelast verbessern und benötigen dafür auch noch weniger Speicher, (iii) die Kosten von Datenänderungen halbieren, und (iv) weniger als 90 % der Daten umverteilen, wenn ein Rechenknoten zum Cluster hinzugefügt wird. Wichtig ist, dass wir die entsprechenden ILP-basierten heuristischen Lösungen innerhalb weniger Sekunden berechnen können. Schließlich demonstrieren wir, dass die Ideen von unseren ILP-basierten Heuristiken auch auf das Indexauswahlproblem anwendbar sind.

Acknowledgements

I am deeply grateful for the forming and enjoyable time at the Enterprise Platform and Integration Concepts (EPIC) research group and, generally, the Hasso Plattner Institute (HPI) and the University of Potsdam. I thank *everybody who helped me* finishing this thesis and want to emphasize my gratefulness to a few people.

First, I sincerely thank my supervisor Prof. Plattner and his chair representatives, Dr. Michael Perscheid and Dr. Matthias Uflacker, for their continuous support, inspirational vision, guidance, and the freedom to pursue my research interests. I also thank SAP, the European Union, and the HPI research school for funding my research.

Second, I thank my colleagues from the EPIC research group. I am particularly grateful to Dr. Rainer Schlosser, my closest scientific collaborator and mentor. Special thanks go to my long-time office mates, Günter Hesse, Christoph Matthies, and Jan Koßmann, for our overly pleasant time in the "normal office" and after work. In this context, I also thank Keven Richly and Martin Boissier for the numerous discussions, conversations, and laughter. Thank you for our collaboration and friendship!

Finally, I thank my family and friends beyond the EPIC group. Special thanks go to my sister and her family, and my parents for their tireless support and love. I am infinitely grateful to Ina Halfpap and our kids, Fynn and Mika. I love you!

Contents

1

Introduction

Almost all programming can be viewed as an exercise in caching.

Terje Mathisen

Column-oriented, in-memory database systems, such as SAP HANA [74] or Hyrise [66], are well-suited for mixed workloads, consisting of transactional and analytical queries [177, 204]. Using these database systems in enterprises has led to the development of new interactive applications [178]. These applications attract a growing number of users, who submit increasingly complex queries. This analytical load can easily saturate single-node database systems.

The increasing demand for processing capabilities can be managed by scale-out approaches, using additional database machines [62, 170]. Scale-out options differ in terms of how the data is stored and how queries are processed. Suitable scale-out options depend on the queried data set (e.g., its overall storage consumption) and the workload (e.g., the read-write ratio). Structured data sets (e.g., the operational data in enterprise systems) can often be stored on a single node [178]. In addition, analyses of real-world workloads show that read-only queries account for the largest workload share [27, 125]. Scaling read-only queries is relatively simple because we can execute them on replica nodes (storing snapshots) of the primary node without violating transactional consistency: The primary node processes all database transactions and propagates committed data modifications to the replica nodes [90]. Such an approach is called *primary replication* and is a common scale-out option for single-node database systems in practice. All major relational database systems, e.g., SAP HANA, Oracle, IBM DB2, Microsoft SQL Server, PostgreSQL, and MySQL, support primary replication [105, 112, 133, 200, 209, 231]. Therefore, all queries are processed locally at the chosen replica.

Using a naive approach for primary replication, called full replication, queries are distributed among nodes independently of the accessed data. This approach has several drawbacks: First, all nodes have to store or potentially load all data. Second, 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.

1.1. Query-Driven Workload Distribution

To tackle these problems, query-driven workload distribution balances the load based on the accessed data. In this thesis, we focus on partially replicated database clusters [102, 181] for a cost-efficient scale-out of single-node database systems as one specific use case for query-driven workload distribution. The distribution of queries based on the accessed data is a generally beneficial concept for optimizing *cost/performance* [144], e.g., (i) in caching architectures, such as data marts [30] or mid-tier caches [131, 146], and (ii) for operator placement in distributed database systems [54, 223]. The core idea of query-driven workload distribution is optimizing the data allocation (e.g., storing less data to reduce hardware costs, or improving data caching to increase the performance) while the load can be evenly balanced, which is crucial for resource utilization and, thus, scalability.

Partially Replicated Database Clusters

In contrast to full replication, partial replication [102, 181] lowers a cluster's overall memory consumption and is, therefore, a memory-efficient implementation of primary replication. Partial replication consists of two steps, which are typically separated from each other to better deal with the problem complexity [169, 181]. 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. The stored data of partial replicas do not have to be disjoint: For example, to execute the same query locally at two different replicas, the replicas must store the same data fragments.

Figure 1.1 shows an exemplary allocation input (left) and allocation with four nodes (right). The input consists of a partitioned database with ten fragments and a workload with five queries. For each query, the workload share (e.g., 10% for query q_1) and the accessed fragments (e.g., the four fragments 1, 2, 3, and 4 for query q_1) are given. In this introductory example, we assume read-only queries so that load for fragment modifications caused by queries does not occur. The allocation is a distribution of the queries' workload shares to the nodes (e.g., the workload shares of the queries q_1 and q_2 are entirely assigned to node 3). The query distribution defines/implies the fragment allocation to the nodes: As q_1 and q_2 access the fragments 1 - 6, these fragments must be stored at node 3. In contrast to a cluster with four full replicas (storing overall $4 \cdot 10 = 40$ fragments), the partially replicated database cluster stores only 14 ($= 3 + 4 + 6 + 1$) fragments, visualized as non-transparent fragments. The shown allocation has the property of distributing the overall workload evenly over the four nodes, i.e., 25% (1/4) of the overall load is assigned to each node. Such allocations are of special interest because they enable a linear scaling of the database load with the number of nodes.

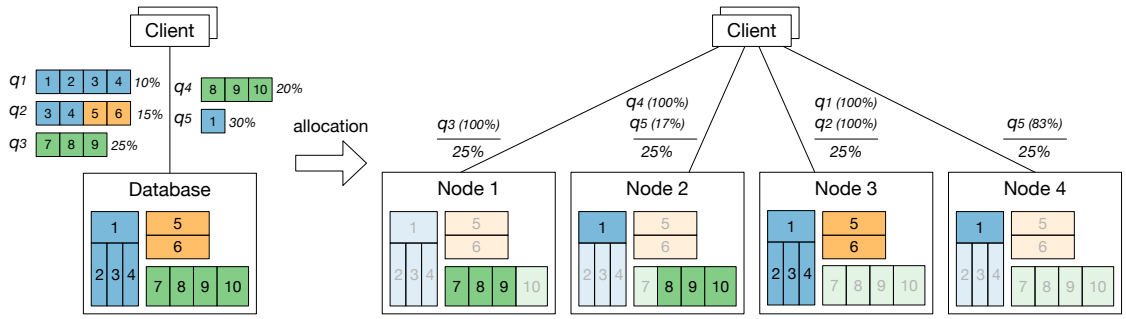


Figure 1.1.: Query-driven workload distribution for a partially replicated database cluster: The left-hand side visualizes the model input, i.e., a partitioned database and five queries characterized by their accessed fragments and workload shares. The objective is to minimize the replication cluster's overall memory consumption while balancing the load across the four nodes. Processing a query requires storing its accessed fragments. The right-hand side illustrates an allocation with 14 fragments overall and an even workload distribution, i.e., 25% of the workload share assigned to each node. Transparent fragments visualize the memory savings per partial replica. (based on [98])

Advantages. Partially replicated database clusters have several advantages compared to full replication. As partial replicas only store subsets of the data, partial replication lowers the cluster's overall memory consumption. Further, it can reduce the replica synchronization costs because modifications occur only for stored fragment subsets. We can also integrate new (partial) nodes more quickly into the cluster because less data has to be loaded or reallocated. Because queries are distributed among replica nodes based on the accessed data, partial replication improves caching. Real-world workload analyses show that the share of frequently accessed data is small [177, 178], resulting in possibly high memory savings for partial replication compared to full replication.

Challenges. As the basis of partial replication, we can use known partitioning approaches, e.g., using single columns (or column groups) for vertical, and range partitioning for horizontal partitioning (although finding the best partitioning criteria is not trivial). In contrast, the allocation step is challenging in theory (many allocation problems are NP-hard) and practice (the number of queries and fragments is high) [169].

Partial replication requires workload knowledge to enable an even load balancing. Calculating efficient fragment allocations that minimize the replicas' memory consumption while evenly balancing the query load is difficult because the two goals contradict each other: On the one hand, even load-balancing requires flexibility to forward queries to nodes with a currently low load. On the other hand, efficient partial replicas aim to store as little data as possible, which lowers the number of

queries that can be processed and, therefore, the flexibility of load-balancing. In particular, Rabl has shown that the calculation of optimal partial allocations is an NP-hard problem [180].

Calculating partial allocations becomes even more challenging when we want to ensure robust performance: When nodes fail or unexpected workloads occur, the load balancing of a partial replication cluster can become skewed because the processable queries per node are limited.

Consider the allocation in Figure 1.1: If node 4 fails, its workload share of query q_5 must be taken over by the remaining nodes 1 - 3. However, with the existing fragment allocation, only node 2 and 3 can take over the load of q_5, which requires fragment 1. Because node 2 and 3 cannot shift any of their query load to node 1, the load balancing becomes skewed. If (instead of node 4) node 1 would fail, query q_3 could no longer be executed by the cluster until fragments are reallocated.

Similar to node failures, unexpected workloads can also lead to skewed load balancing for partial allocations. For example, if the workload share of query q_3 increases unexpectedly, node 1 becomes overloaded because it is the only node that can execute query q_3 and cannot pass any of its assigned load (100% of q_3) to the other nodes. Unexpected queries (e.g., q_6 accessing fragments 7 and 10) could only be executable by the cluster once fragments are reallocated.

In practice, nodes may fail and future workloads are not entirely predictable. Thus, robust allocations that enable an even load balancing despite node failures and workload fluctuations are desirable. Generally, the allocation of additional fragments increases the robustness of the cluster. For example, if we allocate fragment 7 to node 2, query q_3 remains executable if node 1 fails. At the same time, if the workload share of query q_3 increases, node 2 can overtake q_3's load and shift q_1's load to node 3 and 4. Unexpected queries that access possibly unrestricted fragment sets can (only) be executable (without ad-hoc reallocations) if all fragments are stored on a single node (e.g., the primary node).

While the simultaneous optimization of a cluster's robustness and memory efficiency increases the allocation complexity, the calculation time of allocation approaches must remain low to be applicable in practice. This leads to the following allocation goals.

Allocation Goals. We summarize three goals for calculating partial allocations:

1. Flexibility for an even load balancing to ensure **robust performance** of the database system, even if nodes fail or future workloads are uncertain.

2. **Low memory consumption**, e.g., to reduce hardware and replica synchronization costs, improve caching, speed-up reallocations, and save memory for memory-intensive execution algorithms.

3. A **short calculation time** to be also applicable to larger problems in practice, for the initial allocation and if a reallocation is necessary.

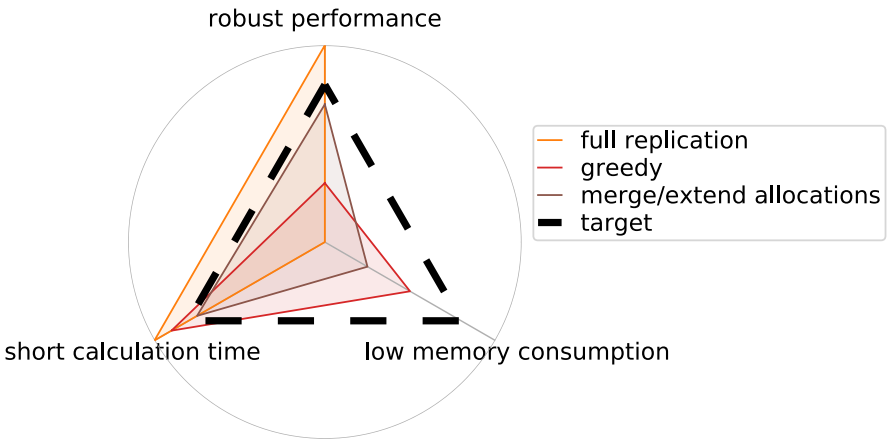


Figure 1.2.: Allocation goals and how they are met by existing approaches: no approach satisfies all three dimensions.

Because the calculation of optimal partial allocations is NP-hard [180], heuristic approaches have to be used for large problem sizes. As optimization goals and constraints for specific allocation problems differ, heuristics are often tied to specific formulations of the allocation problem [170].

For balancing the workload in partially replicated database clusters, Rabl and Jacobsen propose a greedy state-of-the-art approach [181], allocating queries to nodes one after another, thus having a short computation time. Allocations can be greedily extended with fragments to guarantee the executability of queries if nodes fail. However, this approach does not guarantee an even load balancing in failure cases. To ensure an even load balancing in case of workload fluctuations, Rabl and Jacobsen propose to calculate separate allocations for multiple representative workload distributions and merge all of these allocations resulting in a robust allocation. However, because entire nodes are merged, potential to optimize the memory consumption is lost. The overall calculation time increases with the number of workload scenarios being considered. We find that existing heuristics for partially replicated database clusters have different strengths and weaknesses, but none of them fully satisfies the three allocation goals previously listed.

1.2. Allocation Approach

This thesis's primary goal is to develop approaches for quickly calculating memory-efficient allocations for partially replicated database clusters that ensure robust performance despite potential node failures and workload uncertainty. Figure 1.2 visualizes this goal related to full replication and the partial replication approach of Rabl and Jacobsen [181]: Compared to Rabl and Jacobsen's approach, we want to calculate allocations with a better combination of robust performance and low memory consumption. Therefore, we are willing to use slightly higher calculation

times, i.e., a couple of seconds. Note, for our specific problem, there is no need to calculate allocations instantly because the actual data (re)allocations require time, too: Overall, it might be better/faster to calculate longer when we, in turn, save (re)allocation costs. As priorities for the allocation goals might differ among users, we want to be able to balance the three target dimensions of the problem flexibly.

To achieve our goals, it would be possible to improve and extend existing approaches. However, we find it difficult to determine how and where to adapt these *imperative* approaches to simultaneously optimize *robustness and memory consumption*, which contradict each other. For this reason, this thesis presents novel *declarative* allocation approaches for partially replicated database clusters using *integer linear programming (ILP)*. ILP is an common approach for solving assignment problems, also in the context of physical database design [26, 57, 172, 184, 225]. ILP belongs to the class of mathematical optimization (also called mathematical programming) and is a declarative approach. We have to model the allocation problem as a linear objective function and suitable linear constraints. Highly optimized solvers are then used to calculate the solution based on the supplied input data. When using mathematical optimization, the main challenge is to control the model's complexity by a suitable selection of the objective function and constraints. Using *linear* programming restricts the model's objective and constraints to be in linear form, but solvers can compute allocations more quickly than for more general models, e.g., using quadratic objectives or constraints. However, solving ILP models in their general form, in which variables are restricted to integer values, is NP-hard. This is why, related work (e.g., [26, 181]) often uses ILP only for small problem instances and falls back to simple (e.g., greedy) heuristics, which may work well but can lose optimization potential, as our evaluation shows. Instead, *we use ILP to construct powerful heuristics*: In particular, we use (i) a decomposition (i.e., divide-and-conquer) approach, (ii) query clustering, and (iii) solver relaxation techniques to reduce the complexity of individual integer linear programs.

Thesis Statement: *Allocation approaches based on integer linear programming can quickly calculate allocations for partially replicated database clusters with a better mix of robust performance and memory efficiency than state-of-the-art (greedy) approaches.*

We evaluate our optimal and ILP-based heuristic programming models (i) numerically by calculating allocations for different workloads (i.e., the TPC-H and TPC-DS benchmarks as well as a real-world accounting workload) and (ii) their end-to-end performance using a cluster of commercial columnar in-memory database systems.

Solution Concepts

Our ILP models are tailored to calculate allocations for partially replicated database clusters. Nevertheless, using the declarative nature of mathematical optimization, the following individual solution concepts can also be applied to solve other problems in the field of physical database design.

- **Heuristics using mathematical optimization.** The complexity of programming models grows with the number of variables and constraints. However, resources do not have to be allocated in a single step, i.e., with a single mathematical program. Instead, we can decompose large problems into a tree of multiple easier-to-solve problems that preserve the structure of the original problem. Besides, we can also heuristically reduce the input data, e.g., using clustering or pruning.

- **Declarative extension of basic models to ensure robust performance.** Given a basic problem formulation, robustness can be considered by additional rules. Using mathematical optimization, these rules are formulated as constraints. Given the declarative nature of mathematical optimization, robustness constraints can be easily adapted to allocation goals. We find expressing these extensions within a mathematical program is easier in comparison to imperative heuristics, for which it may be difficult to determine how and where to adapt the algorithm without losing unforeseeable traits of optimality.

Application to the Index Selection Problem

We find these two concepts not only beneficial for the data allocation in partially replicated database clusters but also for solving the index selection problem. The selection of indexes is an important task, as indexes are essential for speeding up queries on large databases. Existing index selection approaches have different strengths and weaknesses. On the one hand, approaches based on ILP guarantee optimal solutions but cannot consider all combinations of indexes and broader indexes for larger workloads [123]. On the other hand, many heuristic approaches can find beneficial indexes with many attributes and large index combinations by step-wisely building solutions, e.g., iteratively extending the current solution by appending an attribute to an existing index or adding a new index [196]. However, while the step-wise modifications of the current index selection are often locally optimal, their greedy nature may lose optimization potential.

We demonstrate that we can combine (i) the power of ILP for solving combinatorial problems and (ii) the explorative nature of index selection heuristics for evaluating broad indexes and large combinations: Index selection heuristics use cost estimates to determine their final solution. We use the cost estimates as input of ILP models to find overall better index selections. Further, we present an ILP-based decomposition approach for heuristically solving large index selection problem instances. Finally, instead of a single workload, we investigate the index selection for multiple potential future workload scenarios: Using mathematical programming enables us to extend the optimization function to consider the performance variance of individual workload scenarios for the index selection problem.

1.3. Contributions

In the following, we summarize the primary contributions of this thesis, list the corresponding publications, and shortly describe the thesis author's stake per contribution. Appendix A lists all our publications.

1. **Basic ILP models for calculating optimal and heuristic allocations for partially replicated database clusters**

   We present basic integer linear programming (ILP) models for calculating both optimal and heuristic fragment allocation for partially replicated database clusters. The basic models calculate allocations for read-only workloads. We show how we can scale our optimal ILP model heuristically, using a decomposition approach, input clustering, and solver relaxation techniques. While optimal solutions poorly scale, we can calculate optimized allocations for thousands of query classes and hundreds of fragments in just a few seconds. Our evaluations show that our solutions can almost halve the amount of allocated data compared to state-of-the-art approaches.

   This contribution is based on the following publications:

   - [102] Stefan Halfpap and Rainer Schlosser. Workload-driven fragment allocation for partially replicated databases using linear programming. In Proceedings of the International Conference on Data Engineering (ICDE), pages 1746–1749, 2019.
   - [101] Stefan Halfpap and Rainer Schlosser. A comparison of allocation algorithms for partially replicated databases. In Proceedings of the International Conference on Data Engineering (ICDE), pages 2008–2011, 2019.

   Schlosser and the thesis author contributed equally to the publication [102] and the related demonstration paper [101]. Both authors shared the publications' conceptualization, the ILP models, conducted evaluations, and written text. The thesis author implemented most of the evaluation platform for retrieving model inputs, comparing allocation algorithms, evaluating and visualizing allocation results, and running end-to-end experiments.

2. **Model extensions for considering node failures, workload uncertainty, data modifications, and reallocation costs**

   We present extensions of our basic models for increasing the robustness of allocations to compensate for node failures and workload uncertainty, and considering data modifications and reallocations. The trade-off between memory efficiency, robustness, and a short computation time can be balanced in a targeted way. Compared to existing approaches, our solutions can improve the throughput in case of node failures and workload uncertainty while using even less memory, halve the amount of data modifications, and reallocate less

than 90% data when adding a node to the cluster. Thereby, the calculation of corresponding ILP-based heuristic solutions takes only a few seconds.

This contribution is based on the following publications:

- [104] Stefan Halfpap and Rainer Schlosser. Memory-efficient database fragment allocation for robust load balancing when nodes fail. In Proceedings of the International Conference on Data Engineering (ICDE), pages 1811–1816, 2021.
- [195] Rainer Schlosser and Stefan Halfpap. Robust and memory-efficient database fragment allocation for large and uncertain database workloads. In Proceedings of the International Conference on Extending Database Technology (EDBT), pages 367–372, 2021.
- [103] Stefan Halfpap and Rainer Schlosser. Exploration of dynamic query-based load balancing for partially replicated database systems with node failures. In Proceedings of the International Conference on Information and Knowledge Management (CIKM), pages 3409–3412, 2020.
- [98] Stefan Halfpap. Efficient scale-out using query-driven workload distribution and fragment allocation. In Proceedings of the VLDB PhD Workshop, 2019.
- Stefan Halfpap and Rainer Schlosser. Fragment allocations for partially replicated databases considering data modifications and changing workloads. Under submission, 12 pages.

Schlosser and the thesis author contributed equally to the publications [104, 195], particularly its conceptualization, the ILP models, numerical evaluations, and text. The thesis author implemented the state-of-the-art approaches, and conducted the end-to-end evaluations as well as workload analyses.

Further, the thesis author wrote the first draft of the related demonstration paper [103] and implemented the application. Schlosser supported the conceptualization and revised the paper.

The publication [98], which introduces our model extensions, is a contribution to the VLDB PhD workshop.

For this thesis, we reconducted the entire evaluation for a commercial columnar in-memory database system cluster and extended our model to consider data modification costs and reallocations. The companion paper, for which Rainer Schlosser supported the conceptualization and revised the first draft, is under submission.

3. **Optimal and heuristic ILP-based approaches for selecting indexes**

   To highlight our approaches' flexibility and effectiveness, we demonstrate the applicability of the solution concepts to the index selection problem: We present an optimal ILP model and ILP-based heuristics for selecting database

indexes. By lowering the computation time through our heuristic candidate selection and decomposition approach, we can also solve more complex index selection formulations with wider indexes, more indexes per query, or even consider the performance variance of multiple workload scenarios.

This contribution is based on the following publications:

- [123] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. Magic mirror in my hand, which is the best in the land? An experimental evaluation of index selection algorithms. Proceedings of the VLDB Endowment, 13(11): 2382–2395, 2020.
- [194] Rainer Schlosser and Stefan Halfpap. A decomposition approach for risk-averse index selection. In Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM), pages 16:1–16:4, 2020.
- [100] Stefan Halfpap. Hybrid index selection using integer linear programming based on cached cost estimates of heuristic approaches. In Proceedings of the International Workshop on Simplicity in Management of Data (SiMoD), pages 5:1–5:4, 2023.

For publication [123], the primary authors, Kossmann and the thesis author, contributed equally, in particular to the paper's conceptualization and original draft. The thesis author initiated the research survey, implemented and revised a large share of the evaluation platform (e.g., the ILP approach "*CoPhy*", *AutoAdmin*, *Anytime (DTA)*, *Relaxation*, and the cost evaluation), classified the evaluated algorithms with regard to the underlying approaches, and discussed challenges for commercial index selection tools. Jankrift implemented a prototypical version of the evaluation platform. Jankrift and Schlosser contributed to the paper's concept, and improved its material and presentation.

The thesis author co-authered [194], and contributed to the paper's concept, implementation, evaluation, and text. Specifically, the author co-designed the ILP model, implemented the cost evaluation, and derived the model input. Schlosser initiated the robustness extension and conducted the evaluation.

In this thesis, we focus on a deeper evaluation of ILP-based index selection algorithms compared to our previous survey [123]. Further, this thesis generalizes the decomposition approach [194] for multiple indexes per query and proposes a hybrid index selection approach using ILP based on cached cost estimates of heuristic approaches. The hybrid approach is described in a single-author paper [100].

1.4. Structure of Thesis

The remainder of this thesis is structured as follows: In Chapter 2, we discuss database system scalability and replication approaches. Then, we describe the work-

load distribution problem for partially replicated database clusters in Chapter 3. In Chapter 4, we review related work. We present our allocation approaches in Chapter 5. In Chapter 6, we evaluate our allocation models numerically and in end-to-end experiments. In Chapter 7, we show how we can apply our ILP-based heuristics and robustness extensions to the index selection problem. Finally, we conclude this thesis with a summary and discussion of future work in Chapter 8.

2

Background

Everything is one - except for the zero.

Wau Holland

Partial replication for database clusters is one specific scale-out approach for database systems. In this chapter, we give a more general overview of scalability (Section 2.1) and replication approaches (Section 2.2) for database systems. The covered aspects are selected to (i) position the addressed system architecture in the field of diverse scaling approaches for database systems and (ii) understand it to derive suitable data allocation approaches.

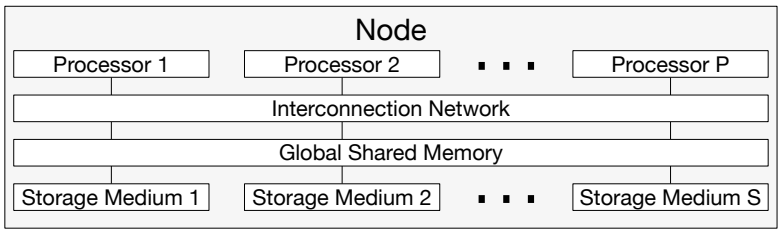
2.1. Database System Scalability

This section presents approaches for scaling database systems with a focus on database clusters. First, we give an overview of scaling approaches (Section 2.1.1). The overview includes an introduction of parallel architectures (i.e., shared-memory, shared-disk, and shared-nothing) and the terms scale-up and scale-out. Following, we describe and compare the three parallel architectures for scaling database systems in Section 2.1.2. Finally, we explain the characteristics of a database cluster, which is a specific form of a scale-out database system (Section 2.1.3).

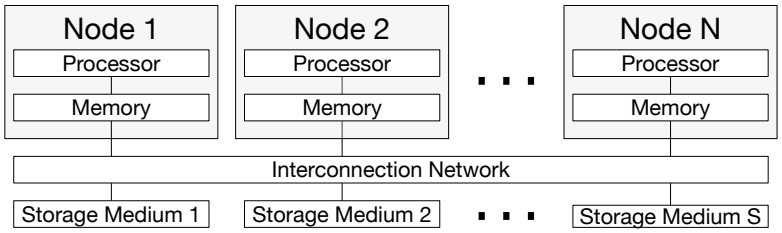
2.1.1. Overview

Scalability is the property of a system to cope with an increasing amount of workload (and data) by adding (hardware) resources [51], e.g., processors with cores and local caches, main memory, and storage. System architectures for scaling database systems (and in general for parallel data processing) can be classified by the way how the processors are interconnected [170] and, thus, share hardware resources. The shared resources affect how processors can coordinate with each other and how they can access stored data.

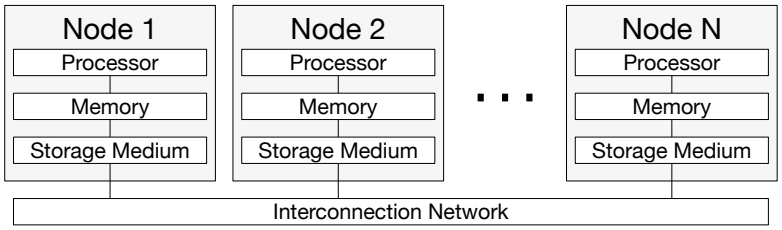
Traditionally, parallel system architectures are divided into three classes [17, 62]: (i) *shared-memory* (see Figure 2.1a), (ii) *shared-disk* (see Figure 2.1b), and



(a) Shared-memory architecture.



(b) Shared-disk architecture.



(c) Shared-nothing architecture.

Figure 2.1.: Parallel system architectures for scaling. (based on [62])

(iii) *shared-nothing* (see Figure 2.1c). Despite the increased and increasing heterogeneity of hardware resources (e.g., by the usage of non-volatile [8, 9] and disaggregated memory [121, 238] in database systems), this classification is still applicable and used. In practice, mixtures of these architectures are common, e.g., a shared-nothing architecture with nodes having multiple processors with shared memory.

Using this classification, the shared-disk and shared-nothing architecture follow a *scale-out* approach [170], i.e., they use an increasing number of loosely coupled processor nodes to scale. In contrast, the shared-memory architecture follows a *scale-up* approach [170], i.e., it scales with more powerful, tightly coupled hardware at a single node (e.g., using processors with more cores or higher clock rates).

2.1.2. Parallel Database System Architectures

The three parallel system architectures have different advantages and disadvantages with regard to their performance (including extensibility/scalability), price, and

resilience (in particular availability) [17]. This section describes and compares the three parallel architectures under these aspects for scaling database systems.

Shared-Memory

In a shared-memory (or sometimes called shared-everything) architecture, all processors can access the entire main memory and storage (see Figure 2.1a). One major advantage of this architecture is the simplicity of programming because all processors run under a single operating system and can communicate quickly via main memory.

Further, load balancing is simple [220]: Each processor can access the entire database and process each query. Naturally, processors can process queries in parallel for *inter-query parallelism*. Due to the quick communication via main memory, processors can also efficiently share or steal work for processing single queries in parallel for *intra-query parallelism*.

Originally, the processors' access to the entire main memory was uniform via the same interconnect (see Figure 2.1a) with the same latency and bandwidth. When all processors share the same interconnect to access the main memory, this *uniform memory access* (UMA) can become a performance bottleneck [214], in particular with an increasing number of processors [65].

Non-uniform memory access (NUMA) architectures try to overcome this limitation but keep the shared-memory view of all processors [87, 137]. In this subclass of shared-memory architectures, each processor has its own local main memory. Processors can access their local memory more efficiently than the memory of the other processors. Overall, the differences in local and remote memory access times are comparably small compared to the differences between memory and disk access times. Nevertheless, the allocation of processing load and data structures to processors and memory in a NUMA system influences the performance [29, 58, 117].

With an increasing number of processors in a shared-memory system, the interconnect to ensure cache coherency efficiently becomes more complex and, thus, expensive [220]. Further, with a growing number of processors and their speed, the probability of conflicting accesses, which degrade the performance, increases [220]. Another disadvantage is the limited scalability of main memory on a single node due to physical limitations for DRAM density and, thus, capacity [115, 148].

Finally, shared-memory systems offer limited resilience [220], e.g., a crashing operating system affects the single node and, thus, the entire system.

As the scaling of shared-memory systems is limited, increasingly expensive, and not particularly resilient, *scale-out* systems that use a shared-disk or shared-nothing architecture have emerged. The idea is to use multiple loosely-coupled processor nodes with their own memory that coordinate via storage or network.

Shared-Disk

In a shared-disk architecture, processor nodes do not share main memory (see Figure 2.1b). Thus, they must coordinate via shared storage or network. The two traditional main technologies to share disks in a cluster are network-attached storage (NAS) and storage-area network (SAN) [170]: NAS offers file-based disk access via a distributed file system protocol, e.g., Network File System (NFS). In contrast, SAN access is block-oriented and typically via a dedicated storage network. SAN is, thus, faster but more expensive than NAS. New shared-disk systems [16, 226] use cloud object stores as "virtually limitless" shared storage.

Compared to a shared-memory architecture, shared-disk increases a system's resilience: Increased resiliency for computing is achieved by multiple independent processor nodes [219], which run their own operating system instances. Data and hardware redundancy can be implemented in the shared storage, e.g., by using multiple independent disks. To avoid a performance bottleneck for accessing the shared storage, shared-disk systems require an efficient or special interconnection network for the storage [221].

Compared to a shared-memory architecture, the coordination between processors is slower, as they are located at different nodes and cannot communicate via main memory. Further, compute nodes need to pull data from the slower shared storage for processing. Compute nodes may have local storage for caching [54]. However, the potentially increased latency with its higher potential for transactional conflicts and lower throughput puts shared-disk systems at a disadvantage for OLTP [12]. For OLAP, the higher latency is no deciding factor because the latency is fast enough for interactive analytics: For OLAP, efficient usage of the parallel hardware is important, e.g., load balancing across the processor nodes. For shared-disk systems, load balancing is easy, as nodes can access the entire data set on shared storage. This aspect is beneficial for skewed workloads with non-uniform access frequencies over the dataset.

Compared to shared-nothing architecture, a big advantage of shared-disk is that the compute layer and storage layer can be scaled independently based on the demand. For example, we can increase storage while simultaneously shutting down idle compute resources. This flexibility has helped shared-disk architectures to a new renaissance for data management in the cloud, where elasticity is a key requirement [12, 212]. Before, shared-nothing architectures have long been the primary design choice for scale-out database systems [62, 211].

Shared-Nothing

In a shared-nothing architecture, processors have their own main memory and storage and must coordinate via the network (see Figure 2.1c). In theory, this architecture is the most scalable, as there is no central component (The interconnection network may be the Internet.). This scalability enables to build huge, even global-scale, data management systems based on cheap hardware, as, for example, Google

has shown [13]. The nodes of a shared-nothing system can be more tightly coupled via a faster and more reliable local area network or loosely via a wide area network. Depending on the coupling, sometimes parallel and distributed database systems are differentiated.

Compared to shared-disk systems, shared-nothing can be said to be cheaper, as they require no particularly efficient and, thus, expensive interconnection network [221]. In addition, processing nodes store the data locally. Hence, there is no (potentially) increased data access latency for loading data from the shared storage. This lower latency enables shared-nothing systems to have a higher efficiency and performance than shared-disk systems. However, the performance depends on the load balancing, which is harder to achieve for a shared-nothing system: Processors can only access their local storage. Data exchange via the shared interconnection network may be limited compared to shared storage and the more tightly coupled nodes of a shared-disk system [185, 236]. Hence, in contrast to a shared-disk architecture, we must decide how to distribute the data set among the nodes in advance [221]. This distribution limits how processing can be balanced. Hence, the performance of a shared-nothing system may degrade significantly under skew [170].

Compared to a shared-disk architecture, the data distribution to local nodes makes implementing elasticity more challenging [221]. For example, when new nodes are added to the system, we must reconsider the load balancing, which may require reorganizing the nodes' data. Further, processing capabilities and storage capacity cannot be scaled independently: Even if only compute is the bottleneck, the added node affects the system's data organization.

In general, the design of a shared-nothing system is more complex compared to shared-disk because the database system functionality has to be implemented via multiple nodes that share only the interconnection network [221]. There are also more design decisions (e.g., how to distribute data over the nodes including data replication and how to implement query processing). Since the first shared-nothing database systems in the 1980s (e.g., Bubba [31, 47] and Gamma [60, 61]), diverse scale-out systems have been developed.

2.1.3. Database Cluster

A database cluster is a specific form of a scale-out database system [170]. It consists of multiple nodes that run autonomous off-the-shelf database systems [170, 186, 187, 188], which are connected by a middleware to act as a single system. In contrast, native parallel and distributed database systems have all nodes under full control. Most database clusters use a shared-nothing architecture, which supports database autonomy with its independent storage better than a shared-disk architecture [169].

The middleware functionality of a database cluster can be integrated into the autonomous database system or be implemented as an external system. The middleware controls the data distribution across nodes, query processing, including transaction handling and load balancing, and fault tolerance [39].

Database clusters can support inter-query and intra-query parallelism: For inter-query parallelism, queries are routed to different nodes and processed in parallel. Intra-query parallelism is more difficult to achieve because the autonomous database systems are (usually) not cluster-aware [170]. Therefore, the middleware must implement the following process: (i) The query is divided into suitable subqueries. (ii) Cluster nodes can then process the subqueries in parallel. (iii) The subresults are finally composed.

Compared to native parallel and distributed database systems, database clusters are often *cheaper* and offer *a simple way to scale out* because they require *no heavy migration* from a single-node database system [170]. For these reasons, cluster databases have been a common architecture for the first large web or e-commerce sites [155]. Today, all major database systems support clustering. However, for the highest performance efficiency, native parallel and distributed scale-out database systems are superior over database clusters because they can integrate distributed database functionality in its core rather than as middleware on top.

2.2. Replication Approaches

In this section, we discuss data replication approaches with a focus on shared-nothing database clusters. First, we give an overview of replication as part of the data distribution process (Section 2.2.1). Following, we cover partial replication (Section 2.2.2) as an optimization for the data distribution. Finally, we classify approaches to synchronize the replicas' data (Section 2.2.3).

2.2.1. Overview

For database systems with a shared-nothing architecture, we have to decide how to distribute the data among the nodes. There are two techniques to distribute data over the nodes: partitioning and replication [170]. The data distribution is coupled to the overall system design, including the query processing across nodes.

Data Partitioning and Replication

Partitioning is the horizontal or vertical splitting of data items (e.g., relational tables) into *data fragments*, the units of replication [170]. Horizontal partitioning splits tables into subsets of rows, e.g., for scanning these subsets in parallel or storing them at separate nodes.

Vertical partitioning splits tables into subsets of columns. It is, for example, used in columnar database systems, such as Hyrise [66], SAP HANA [74], Vectorwise [240], and Vertica [128]. Columnar database systems store individual attributes separately [48] and can, thus, scan them efficiently [149] without requiring indexes [177]. Horizontal and vertical partitioning can be nested to form a hybrid partitioning scheme.

Database replication means that data fragments are duplicated and stored at multiple nodes. The reasons for database replication are increasing (i) data availability and (ii) system performance. (i) When data is stored at multiple nodes, the data is still available when single nodes fail. (ii) When data is stored at multiple nodes, these nodes can access the data and, thus, process queries in parallel.

Database Distribution

The database distribution is a combination of data partitioning and replication. It is tightly coupled to the query processing model, i.e., how the query workload is distributed across the nodes, and transactional consistency is guaranteed. Suitable approaches depend on the data set size, workload (e.g., the ratio of read-only queries and OLAP vs. OLTP), and system requirements. Two common database distribution approaches for shared-nothing systems are *sharding* and *full replication*:

- **Sharding.** If the overall database size is too large to be stored and processed at a single node, it must be partitioned and distributed across nodes. A prevalent approach based on horizontal partitioning is storing individual fragments at different nodes [49, 52, 60, 111, 227]. In this context, horizontal partitioning is commonly called *sharding*. The single fragments are called shards. In a sharded system, nodes store and execute (sub)queries on individual shards.

  Sharding can be combined with replication: shards are replicated and stored at different nodes to increase their availability. The number of replicas can be adjusted depending on the desired availability.

- **Full replication.** If the database fits on a single node, we can replicate and store the entire database at each node. All nodes can access the entire database and can process queries in parallel. This approach is called full replication and is common for cluster database systems [181].

  If a simple scaling approach is needed, we can use full replication without intra-query parallelism across nodes, i.e., replica nodes process individual queries locally. This approach requires no sophisticated middleware for distributed query processing and is, thus, comparably easy to implement and use in practice. In contrast, local query processing limits the speedup for individual (highly processing-intensive) queries.

  Read-only replication. Analyses of enterprise workloads have shown that most queries are read-only [27, 75, 125]. Further, modern database systems, such as SAP HANA [74], Hyper [113], LeanStore [136], and Umbra [163], can achieve high transaction rates with a single node. Combing both facts makes read-only replication a reasonable scale-out approach [158, 175, 178, 199]. When using read-only replication, a single (powerful) *primary node* handles all data modifying queries (i.e., inserts, deletes, and updates), particularly the entire

transaction processing. Handling transactions at a single node has the advantage of not needing cross-node coordination, which can limit the overall transactional throughput if many conflicts occur. Replica nodes are used for read-only queries, mainly to take over heavy-load analytical queries. Read-only replication is, thus, a suitable choice for enterprise computing, which requires consistency and scalability on mixed workloads [178]. However, the scalability is limited, i.e., the suitability depends on the read-only workload share. In Faust et al.'s analysis of a modern financial system without materialized aggregates [75], the accumulated execution time of read-only queries (i.e., selects) was 98%.

Divergent Design Tuning. When duplicating data, we can also store this data using different divergent designs, e.g., different partitionings for better cache performance [2, 183] or different auxiliary structures to optimize individual query subsets [46, 166]. There are diverse auxiliary structures for tuning, including index [123] variants (e.g., B-trees, hash maps, or bitmaps), pruning filters [4, 23, 89], materialized views [93, 156], caches (managed by the database system) [68]. For example, replicas could store highly specific indices, such as full-text search indices, which are not created on the primary node for capacity reasons. In return, the system must take care of an optimized query routing to nodes that can efficiently process the query and the overall load balancing.

Further, we can also use replica nodes to extend the system's functionality by running different database systems on the replicas. Different database systems could support additional operators (e.g., window functions or the skyline operator) or implement different data models (e.g., the graph model) than the primary node [176].

Transparency of Database Distribution. An important aspect of shared-nothing database systems is the transparency degree of the data distribution, which assesses how much users must care about distributing the data and processing queries [170]. When full transparency is provided, the database system or middleware handles all query distribution and processing aspects, including distributing the data, splitting queries into subqueries for separate nodes, and synchronizing data replicas. When only limited transparency is provided, users must take over parts of this work. For example, they may have to (re)distribute data or route (sub)queries to nodes.

2.2.2. Partial Replication

In Section 2.2.1, we introduced the idea of divergent design tuning when duplicating all data. Another approach for optimizing the data distribution across nodes is partial replication [170]. When using partial replication, the number of replicas per data fragment within the cluster may differ. The idea of this approach is duplicating the data based on the access frequency (and availability goals). Typically,

the majority of (OLTP and OLAP) queries access only a small subset of the overall database [27, 79, 138, 178].

Partial replication has the following advantages: Storing less data reduces (i) hardware costs or (ii) provides more space for efficient data processing. (i) We can use cheaper nodes with less storage. (ii) We can use the saved storage for static auxiliary structures (e.g., to use additional secondary indices). For in-memory database systems, the saved storage (main memory) can also be used for memory-intensive dynamic data structures that may be advantageous for query processing but would be too large without replica optimizations.

Further, storing less data reduces the synchronization costs because replicas must synchronize relevant/fewer data changes. Less synchronization costs lead to more time for query processing. Overall less synchronization costs, in turn, enable smaller hardware configurations, e.g., using fewer replica nodes. Finally, storing less data improves caching and the performance because a larger (relative) data share fits into the (different levels of) caches.

The concept of partial replication is suitable (i) for sharding and (ii) as optimization for full replication. (i) For sharding, we can adjust the number of replicas per data fragment based on the access frequency [151]. (ii) As optimization for full replication when queries are processed locally, partial replicas store only subsets of the data [181]. Partial replicas can process subsets of queries, particularly frequent processing-intensive analytical queries while requiring less storage than a full copy of the database.

2.2.3. Replica Synchronization

When data is duplicated and stored at different nodes, each *logical data item* has a number of *physical copies* in the system. Hence, we must decide how physical copies are synchronized. These design decisions include (i) where we can issue data-altering transactions (i.e., on a single primary node, where the primary copy is located, or anywhere) [170], (ii) when we propagate transactional changes to the other physical copies (i.e., eagerly or lazily [90]), and (iii) the kinds of information (i.e., logical or physical) to update physical copies. In the following, we discuss these independent design decisions, which can be used to classify replication approaches.

Single-Primary vs. Primary-Copy vs. No-Primary

We can differentiate where to issue data-altering transactions for individual logical data items [90, 170]. The *single-primary* approach allows them only on a (single) dedicated node, called the primary node. The data changes are then propagated to the other nodes, called replicas or secondaries.

Primary-copy, also called *multi-primary*, allows data-altering transactions for different logical data items at different nodes, but all updates of specific logical data items must be issued at the same node [41]. We can summarize both approaches

(i.e., single-primary and primary-copy) as *centralized approaches* [170]: For all logical data items, there is a dedicated central physical copy.

In contrast, for a *distributed* [170] or so-called *update-everywhere* approach, there is no primary copy of a data item and, thus, no primary node.

The used approach influences the required coordination and scalability. Centralized approaches avoid coordination between nodes for transactions that affect single data items. Single-primary avoids inter-node coordination for transaction processing completely, but the transaction processing cannot scale beyond one node.

In contrast to single-primary, primary-copy can distribute the transaction processing of individual logical items to different nodes and is, thus, more scalable in theory. However, primary-copy requires inter-node coordination for transactions that update multiple data items with different primary nodes, which may slow down the overall transaction processing.

Using an update-everywhere approach, we can theoretically balance the load best, but the required coordination and danger of transactional conflicts are the highest because different physical copies of a data item can be updated at different nodes concurrently. Optimized assignments of primary copies and transactions to nodes can avoid transactional conflicts and increase the performance.

Eager vs. Lazy Synchronization

We can differentiate when data changes are propagated to replicas [90, 170]. Eager replication propagates updates (individually or batched) to all replicas as part of the transaction. When a transaction is committed, it is executed on every replica atomically. Hence, all physical data items in the cluster are in the same state after the end of a transaction. When using eager replication, the performance to keep replicas in sync is important because it directly influences the transactions' latencies.

In contrast, lazy replication postpones the synchronization of replicas, i.e., the propagation of changes to the other nodes is not part of the transaction.

Lazy replication delivers better transaction latencies than eager approaches because it does not wait to return until all nodes are synchronized. In addition, lazy replication can better optimize the communication between primary and replica nodes by batching update information of multiple transactions without sacrificing individual transaction latencies. The disadvantage of lazy replication is that mutually consistency of replicas is not guaranteed and must (if desired) be implemented on application level. In practice, lazy database replication is often used with snapshot isolation as isolation level [59].

Logical vs. Physical Information

The kind of information on how to synchronize replicas can be logical or physical. Logical updates describe data modifications on a higher level, such as SQL statements. Physical updates provide lower-level information regarding the used data

structures, for example, specifying the offsets where to insert or change values. The size of physical update information depends on the amount of changed data, but the speed of replaying it is usually faster than for logical updates (e.g., SQL statements) because the queries do not have to be re-executed [234].

3

The Workload Distribution Problem

> By the time you're done with your dissertation, you'll have answered questions no one ever answered (or possibly asked) before.
>
> Peter Bailis

This chapter formalizes our workload distribution problem for partially replicated database clusters, which we introduced in Section 1.1. We first describe the basic problem for balancing *read-only* queries among database nodes in Section 3.1. In Section 3.2, we then motivate further aspects (i.e., the calculation time, node failures, workload uncertainty, data modifications, and reallocation costs) that must possibly be considered for allocations in practice.

3.1. Basic Problem

In this section, we describe how we want to distribute a read-only workload among a set of database nodes. We first explain the problem's input in Section 3.1.1, followed by constraints and the solution in Section 3.1.2. Section 3.1.3 concludes this section with an example.

3.1.1. Input

Table 3.1 gives an overview of our problem's input parameters, which we explain in the following more detailedly.

We assume a database consisting of a number of disjoint data fragments/partitions N, the units of replication. The fragmentation scheme (i.e., whether the database is partitioned horizontally or vertically) and data model (e.g., relational, graph, or object model) are not relevant to the problem. The size of fragment i is denoted by a_i, $i = 1, ..., N$. To derive fragment sizes in practice, database systems like PostgreSQL provide metadata in their database catalog. Further, it is also possible to estimate fragment sizes based on the number and internal physical representation of data records.

Further, we assume a database cluster with K nodes/machines. These nodes store (subsets of) fragments and execute queries. As the read-only query throughput

Table 3.1.: Input parameters.

symbol	description
N	number of fragments $i, i = 1, ..., N$
Q	number of queries $j, j = 1, ..., Q$
K	number of nodes $k, k = 1, ..., K$ (cluster size)
a_i	size of fragment $i, i = 1, ..., N$
q_j	accessed fragments of query j, $j = 1, ..., Q$, a subset of {1,...,N}
f_j	frequency of query $j, j = 1, ..., Q$
c_j	costs of processing query $j, j = 1, ..., Q$
C	total workload costs
V	overall size of all accessed fragments

can be scaled linearly with the number of nodes K, K is determined in practice depending on the expected workload or performance goals.

Finally, we assume a set of Q (classes of) queries j, which are characterized by used fragments $q_j \subseteq \{1, ..., N\}$, $j = 1, ..., Q$. It may be impossible to determine whether a query actually accesses a specific fragment before executing the query. In these cases, the set of fragments must be specified pessimistically, containing all possibly required fragments. Further, unknown ad-hoc queries can be modeled as a query class accessing all fragments, i.e., $\bar{q} = \{1, ..., N\}$. Queries j occur with frequency f_j, $j = 1, ..., Q$. The costs of query j are independent of the executing node k, $k = 1, ..., K$, and denoted by c_j, $j = 1, ..., Q$. Query costs are numerical and can be modeled in several ways. Suitable metrics may differ depending on specific database systems. The complexity of metrics ranges from easy-to-measure and widely applicable metrics, such as the average processing time of a query, to advanced metrics derived by cost models for specific database systems, e.g., considering memory hierarchies and access patterns for main-memory databases [150]. Using the query costs c_j and frequencies f_j, we can derive the overall workload costs

$$C := \sum\nolimits_{j=1,...,Q} f_j \cdot c_j.$$

Using the fragment sizes a_i and actually occurring queries $j = 1, ..., Q : f_j > 0$, we can derive the accessed data size

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

3.1.2. Constraints and Solution

Our problem is a coupled data placement and workload distribution problem: We want to decide (i) on which node to put which data fragments and (ii) which query is executed at which node to which extent. Table 3.2 introduces the decision and solution variables. In the following, we describe the decision variables, constraints, aspects to consider, and solution variables.

Table 3.2.: Decision and solution variables.

symbol	description
$x_{i,k}$	fragment i is allocated to node k: yes (1) / no (0)
$y_{j,k}$	query j can run on node k: yes (1) / no (0)
$z_{j,k}$	query j's workload share assigned to node k: $\in [0,1]$
W	overall size of all allocated fragments (memory consumption)
W/V	an allocation's normalized memory consumption (replication factor)
L	a node's workload limit

Decision Variables

To describe calculated allocations, we use the following decision variables:

- $x_{i,k} \in \{0,1\}$, $i = 1, ..., N$, $k = 1, ..., K$, are allowed to be zero or one, indicating whether fragment i is allocated to node k (1) or not (0).
- $y_{j,k} \in \{0,1\}$, $j = 1, ..., Q$, $k = 1, ..., K$, are allowed to be zero or one, indicating whether query j can run on node k (1) or not (0). The values $y_{j,k}$ can be derived by the allocated fragments $x_{i,k}, j = 1, ..., Q$, $k = 1, ..., K$ $i = 1, ..., N$.
- $z_{j,k} \in [0,1]$, $j = 1, ..., Q$, $k = 1, ..., K$, are allowed to be continuously between zero and one, indicating the workload share of query j executed at node k.

Solution Constraints and Aspects

Several constraints and aspects have to be taken into account:

(a) A query j can only be executed at node k if all relevant fragments q_i are stored at node k. This constraint is implied by our system model.

(b) The workload must be balanced among the nodes to enable scalability (see Section 2.1.2, shared-nothing). Otherwise, a single node would become the cluster's bottleneck and limit the overall query throughput.

(c) The sum of workload shares for a fixed query over all nodes must be one.

(d) The data placement should be optimized: We want to minimize the cluster's overall memory/storage consumption. Minimizing the memory consumption has the following advantages (see Section 2.2.2): (1) Hardware storage costs are reduced. (2) Data synchronization costs are reduced. (3) Data caching is improved.

(e) The data placement problem and the workload distribution problem cannot be decoupled and must be simultaneously solved.

Solution Variables

The following two solution variables can be derived from the decision variables and are our key metrics to assess the solution/allocation:

- The **maximum workload limit L**, $0 \leq L \leq 1$, over all K nodes assesses the load balancing (see (b)). If $L = 1/K$, the load can be evenly balanced among the nodes.

- The **replication factor W/V**, where the total amount of data used

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

  is normalized by the amount of accessed data V, assesses the overall memory consumption (see (d)).

The simultaneous optimization of both metrics is a tradeoff: A better (lower) maximum workload limit requires a worse (higher) replication factor (e.g., full replication enables the optimal limit $L = 1/K$). A better (lower) replication factor leads to a worse (higher) maximum workload limit (e.g., storing all fragments only once at a single node implies the worst-case limit $L = 1$).

In general, we can use a fixed maximum replication factor W/V or a fixed maximum workload limit L as input parameter and optimize the other metric. In this thesis, we focus on allocations with an even load balancing $L = 1/K$, which are usually desired in practice.

3.1.3. Running Example

Figure 3.1 visualizes an exemplary model input and solution. In Figure 3.1, we have a database with $N = 10$ fragments, $Q = 5$ queries, and $K = 4$ nodes. All fragments i have the same size, e.g., $a_i = 1$ GB, $i = 1, ..., 10$.

Executing queries requires storing different subsets of fragments, e.g., query $j = 1$ requires the fragments $q_1 = \{1, 2, 3, 4\}$. Processing query $j = 1$ takes $c_1 = 5$ s (query costs). The relative frequency value of query $j = 1$ with regard to the other queries is $f_1 = 20$. Using the overall workload costs $C = 1000$ s, we can derive the workload share of query $j = 1$ as $\frac{f_1 \cdot c_1}{C} = 10\%$.

In Figure 3.1, allocated fragments are colored normally, e.g., for node $k = 1$, $x_{i,k} = 1$ for $i = 1, 2, 3, 4$. Transparent fragments are not allocated, e.g., for node $k = 1$, $x_{i,k} = 0$ for $i = 5, 6, 7, 8, 9, 10$.

Based on the stored fragments, we can derive all executable queries. For example, node $k = 1$ can execute query $j = 1$ and $j = 5$, i.e., $y_{j,1} = 1, j = 1, 5$. The other queries cannot be executed, i.e., $y_{j,1} = 0, j = 2, 3, 4$.

For the executable queries $j = 1$ and $j = 5$ of node $k = 1$, the assigned workload shares are $z_{1,1} = 1$ and $z_{5,1} = 0.5$. The assigned workload shares determine a node's

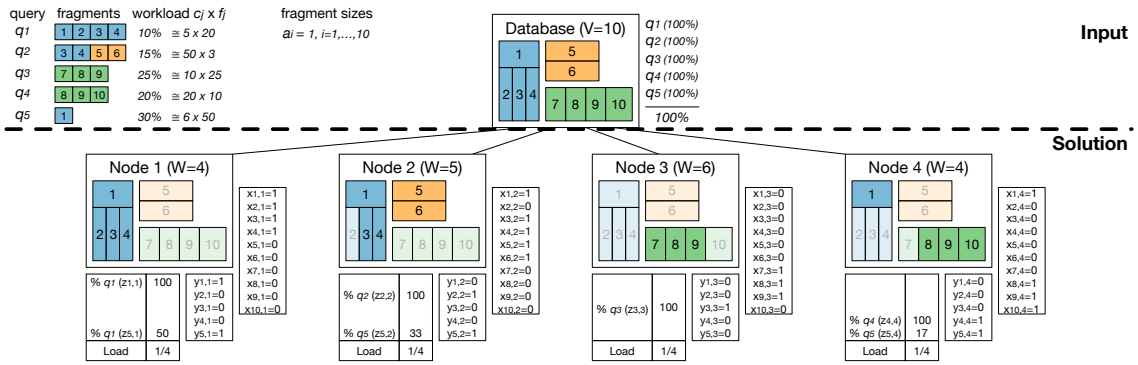


Figure 3.1.: Query-driven workload distribution for a partially replicated database cluster: The upper part visualizes the model input. The database consists of $N = 10$ fragments. $Q = 5$ queries correspond to different workload shares. Processing a query requires storing its accessed fragments. The objective is to minimize the overall memory consumption of the replication cluster while evenly balancing the load among $K = 4$ nodes. The lower part illustrates an example allocation with a total replication factor $W/V = 1.9$ and even workload distribution with $L = 1/4$ of the workload share assigned to each node.

assigned workload share, e.g., for node $k = 1$, $\sum_{j=1,\ldots,Q} z_{j,k} \cdot \frac{f_j \cdot c_j}{C} = 1 \cdot 0.1 + 0 + 0 + 0.5 \cdot 0.3 = 1/4$.

The sum of workload shares for a fixed query over all nodes must be one, e.g., for query $j = 5$, $\sum_{k=1,\ldots,4} z_{j,k} = 0.5 + 0.33 + 0 + 0.17 = 1$.

Overall, the solution in Figure 3.1 satisfies the constraints (a) - (c). The maximum workload limit over all nodes is $L = 1/4$, which is optimal. The replication factor is $W/V = (4 + 5 + 6 + 4)/10 = 1.9$, which is better than for full replication $W/V = 4$, but not (yet) optimal.

3.2. Further Considerations

In practice, allocations must possibly consider further aspects, such as robustness against node failures, robustness against uncertain workloads, data modification costs, and reallocation costs. Further, the calculation time for allocations must be short enough for practical applicability. Considering these factors enables the calculation of robust allocations and adapting them in dynamic setups, in which the workload, data, and system landscape (e.g., whether a node is running or has to be updated) may change over time and, thus, the data allocation can be optimized. Following, we discuss allocation aspects that may be considered in practice.

3.2.1. Calculation Time

Calculating optimized allocations becomes more difficult with an increasing number of fragments, nodes, and queries (see (e)). Calculation times must be low enough for practical applicability. Limits for the calculation time depend on the specific use case. Ideally, an approach allows flexibly trading the allocation's quality (i.e., the workload limit L and replication factor W/V) and calculation time.

After the calculation, an allocation must be deployed, i.e., nodes must physically load the data fragments. As the time for fragment deployment usually increases with the amount of moved data, practitioners are willing to accept higher calculation times for a quicker overall setup. For this reason, allocations do not have to be calculated with interactive calculation times or with basically no overhead (as for CPU caching). Nevertheless, in this thesis, we want to calculate allocations relatively "quickly" (see Section 1.2, thesis statement), below 100 s. Further, we want to investigate optimal solutions and how to trade larger (than 100 s) runtimes to obtain potentially better results.

3.2.2. Node Failures

In practice, nodes might fail. In the presence of node 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 or even require reallocations. A database cluster can support different levels of robustness in case of node failures. Basic robust allocations ensure that (i) each fragment is stored at multiple nodes or (ii) queries can be processed at multiple nodes [181]. Resulting allocations guarantee that (i) all fragments are still available and (ii) all queries are still processable if nodes fail. However, the workload distribution after node failures can be highly skewed. Advanced robust allocations ensure that all queries are still executable in failure cases without overloading single nodes, i.e., in any case of a node failure, it must be possible to balance the workload between all remaining nodes evenly. 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 at which different subsets of queries can be executed. Sophisticated approaches are needed to guarantee balanced workload distributions in the case of node failures, particularly in the standard case of single-node failures.

3.2.3. Workload Uncertainty

Because applications, business processes, and workflows predefine query classes that are sent to the database system, many workloads are predictable to a certain degree. Nevertheless, most workloads are not perfectly predictable. In specific, the query classes' frequencies may fluctuate, e.g., depending on which (part of the) application is currently most used. Further, modeled query costs (e.g., the average processing

time) are inherently imprecise to a certain degree, e.g., because concurrently processed queries (using the same processor caches) influence the runtime.

Data allocations are usually flexible to a certain degree: (i) If a single node can process multiple queries, the workload shares of these queries at the node are flexible. (ii) If queries are (in addition) processable by multiple nodes, workload shares can also be compensated among nodes.

However, the compensation possibilities are limited. If the allocation is only optimized for memory consumption, only a few queries are usually processable per node (see Figure 3.1, node $k = 3$) and queries are often only executable by a single node (see Figure 3.1, queries $j = 1, 2, 3, 4$). If the actual queries' workload shares deviate too strongly from the modeled one, load balancing may become skewed and the query throughput decreases. To increase the robustness against uncertain workloads, we can allocate additional fragments/queries to nodes. We want to ensure an even workload distribution for multiple possible workload scenarios, which are determined by different query subsets, frequencies, and costs. Using an increasing number of diversified input workloads increases the load balancing flexibility and, thus, also robustness against unknown workload deviations to a certain degree.

3.2.4. Data Modifications

As a result of inserts, updates, or deletes, data (fragments) may change over time. These modifications must be performed at all replicas that store the affected fragments by executing data modifying queries (i.e., logical replication, see Section 2.2.3) or performing only the actual data changes (i.e., physical replication, see Section 2.2.3). Although the workload share of data modifications is often small [27, 125], fragment modifications and synchronization cause additional costs, which must be considered for the allocation to ensure an even load balancing.

3.2.5. Reallocation Costs

Workloads (i.e., query subsets, frequencies, and costs) or fragment sizes may change over time. As a result, a current data allocation may not allow an even workload distribution anymore, or a different allocation may reduce the memory consumption. Of course, robust partial allocations that consider node failures or workload uncertainty can compensate for workload changes to some extent. However, in practice, workloads may change heavily or unexpectedly. In these situations, if a low memory consumption despite heavily changing workloads is desired, fragment reallocations may be preferable.

Further, we may want to elastically add nodes to (or remove nodes from) the cluster. Then, we must decide which queries and fragments to allocate to the new node (or which node to remove), and how to reallocate the existing nodes.

It is possible to calculate a new allocation for the changed input from scratch. However, the new allocation could differ enormously from the current one. Then,

the resulting reallocations would be time-consuming and, thus, costly. To avoid costly reallocations, the current allocation can be taken into account by the allocation algorithm. The challenge is to identify *minimally invasive* reallocations, which reallocate little data but optimize the solution, i.e., the load balancing and the memory consumption.

4

Related Work

> Human beings, who are almost unique in having the ability to learn from the experience of others, are also remarkable for their apparent disinclination to do so.
>
> Douglas Adams

Our allocation problem belongs to the class of assignment problems, which have been studied extensively in related work. There are many variations of assignment problems and solution approaches. Thereby, solution approaches can be tightly coupled to specific problem formulations. But, there are also problem-independent algorithmic approaches for solving assignment problems. Section 4.1 gives an overview of related assignment problems and how they differ. We summarize general optimal and heuristic solution approaches in Section 4.2. In Section 4.3, we then focus on more specific allocation approaches that have been proposed for our workload distribution problem and related problems.

4.1. Assignment Problems

Assignment problems are omnipresent and have many real-life applications, e.g., in production planning, transportation and routing, and, naturally, computer and database systems [165]. Öncan surveys a generalized assignment problem, for which (abstract) *items* have to be assigned to (abstract) *knapsacks* under capacity constraints and an optimization function [165]. For specific problem formulations, items and knapsacks relate to different things. In our problem (see Chapter 3), we have to assign queries and fragments (items) to database nodes/machines (knapsacks). If the items relate to data (e.g., files or fragments), the problem is often named "allocation" instead of "assignment". Naturally, the specific *optimization goal* and the *assignment constraints* are more important than naming items and knapsacks in a particular context. For example, specific assignment problems may differ in the constraints (i) whether items can be arbitrarily split or must be assigned as a whole, (ii) or if the assignment should also optimize the robustness, e.g., in case of potential failures or uncertainty.

Given the constraints and optimization function, we can classify assignment problems. An important classification is whether the decision variables are continuous or

discrete (called integer optimization or programming). Integer problem formulations are more difficult to solve. If only some variables are restricted to integers (and the others are continuous), the problems are called mixed integer programming (MIP) problems. If variables are restricted to zero (0) and one (1) rather than arbitrary integers, problems are named to be binary integer programming (BIP).

Further, optimization problems are classified by what kind of functions (e.g., linear, quadratic, exponential) are used for the constraints and optimization function. More restricted functions (e.g., only linear) enable the application of special/faster solution algorithms. In the field of less restricted problem formulations, convex optimization is an important subclass with more efficient algorithms than the general mathematical optimization.

Many assignment problems are expressed using only linear functions. (Integer) linear optimization, called (integer) linear programming, (I)LP, is, thus, an important subclass. How real-world problems are modeled as formalized problem descriptions may depend on different factors.

Finally, we may classify what information about the solution of an optimization problem we seek in practice. We may want to know (i) the value of the optimal solution, (ii) whether a solution exists that is better than a fixed limit, or (iii) the assignment (i.e., the values of the decision variables) for the optimal solution.

In the following Section 4.1.1, we present (classes of) selected assignment problems with integer programming formulations. Section 4.1.2 summarizes variants and extensions of these problems.

4.1.1. Integer (Linear) Programming Assignment Problems

This section presents selected assignment problems, how they differ from our workload distribution problem, and how they influenced proposed solution approaches. In our selection, we focus on assignment problems in computer science and former historical examples, which inspired the later solution approaches. Öncan's survey [165] lists further assignments problems of a wider field.

- **Warehouse Location Problem.** The placement of warehouses or plants [15, 70, 76, 126] is a historically important assignment problem, for which the underlying abstract activities (e.g., storing and transmitting goods) are similar to later problems in the field of computer science. Thereby, a set of warehouses has to be chosen to supply a set of customers with goods while minimizing (transportation and fixed) costs. Insights when solving this older assignment problem have naturally influenced later related problems: For example, Kuehn et al. not only stated that the size and nonlinearities can make problem instances practically infeasible to be solved optimally, but also emphasized the value of heuristics if solution time is a criterion [126].

- **File Allocation Problem (FAP).** Early data assignment problems in the field of computer science are different variants of the file allocation problem [38, 45, 73, 147, 157]: A number of files must be placed in a network of computers under storage, transmission, and data access constraints. Dowdy et al. showed that specific problem formulations are diverse [64]. The main difference when placing files instead of database relations or fragments (which may be co-accessed) is the independence of single file accesses [5]. However, model abstractions (e.g., simplified modeling of queuing delays [45]) of these early assignment problems have been adapted in later problem formulations. Further, earlier work in the context of data allocation has also shown that solution approaches from other domains can be adapted or applied, e.g., if problems are isomorphic [40, 182].

- **Data Fragment Allocation in Distributed Database Systems.** With the development of scale-out database systems, data fragment allocation in distributed database systems has become an important assignment problem. Early overviews in this field summarized differences from the file allocation problem [6, 139]: The database must be first partitioned into suitable fragments. The access of data in a database system is more complex than in a file system. For example, queries access many different subsets of fragments, and data must be combined. Overall, fragments are not independent, and co-location is an important factor [5, 230]. Further aspects do not have to be considered for the FAP but distributed database systems, e.g., transaction handling and implementing specific replication protocols [170].

  Similar to the FAP, there are many specific formulations of the fragment allocation problem [170], which differ in (i) optimization goals (e.g., performance, costs, and availability) and (ii) constraints based on the (simplified) system assumptions. Özsu and Valduriez present a "relatively general" allocation model how to assign fragments to the nodes of a distributed database management system [170] (see also the previous book edition [169]. In their description, they mention many potentially important aspects (e.g., costs for concurrency control) but do not present the model with every detail.

  Our models share many similarities, for example, (i) minimizing storage costs as (part of the) optimization function, (ii) binary decision variables for allocating fragments to nodes, and (iii) the distinction between accessed and updated fragments.

  Compared to their model, in our problem, accessed fragments have to be stored where the query is executed. This constraint is similar to the processing model of Rabl and Jacobsen [181], whose approach is discussed detailedly in Section 4.3.1.

  Further, we develop a linear model while their processing cost constraints contain the nonlinear `min` function.

Besides their general allocation model, Özsu and Valduriez address solution methods [170]: 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 for specific allocation problems differ, heuristics are often tied to specific formulations of the allocation problem.

4.1.2. Variants of Assignment Problems

In the previous section, we mentioned that not every assignment problem is modeled using integer linear programming. In the following, we summarize important variants of assignment problem formulations [165]:

- **Nonlinear Optimization.** Constraints and the optimization function may contain nonlinearities, e.g., quadratic, exponential, `min`/`max` functions. For example, if we want to optimize (or include the optimization of) the variance (called mean-variance optimization (MVO)) [153, 194] to consider the dispersion (e.g., for a risk-averse optimization), we have to use a quadratic optimization function.

- **Dynamic Assignment Problems.** For *dynamic* assignment problems [120], the order of assignments is relevant, e.g., when tasks occur at a specific time or jobs have a due date. An important dynamic problem in the context of cloud computing is the assignment of virtual machines to machines of large data centers [96, 215, 217, 224].

- **Stochastic Assignment Problems.** Further, there are stochastic [3, 106] problem formulations, for which the problem contains uncertainty, e.g., the cost of assigning a job to a machine, or the probability that a job occurs. Also, future workloads (e.g., the frequency of queries) may be uncertain.

Of course, these variants are independent and can be arbitrarily combined. Further, there is often a difference between real-world problems/processes and how they are modeled to obtain a solution (as we mentioned before).

4.2. General Solution Approaches

Specific allocation problems differ in the optimization goal and constraints. Nevertheless, we can calculate the solution using similar solving techniques, e.g., integer linear programming (or mathematical programming in general) or greedy search. Suitable solving techniques depend not only on the targeted solution quality but also on the problem instance (i.e., problem formulation and input sizes) and limits for the calculation time, e.g., whether the result must be computed within a few

seconds for an interactive application or not. Often, it is possible to trade solution quality and computation effort, i.e., we can find better solutions with more computation time or resources. In the following, we survey different optimal (Section 4.2.1) and heuristic (Section 4.2.2) solution approaches. Besides these solution approaches, there are also (arguably) more general algorithmic paradigms/patterns, e.g., divide-and-conquer and dynamic programming. Divide-and-conquer algorithms (recursively) break a problem into subproblems, solve the subproblems individually, and combine the subproblems' solutions into the overall solution [50]. Dynamic programming algorithms (recursively) break a problem into subproblems and reduce the overall calculation time by storing and reusing subproblems' solutions [50].

4.2.1. Optimal

Exhaustive Search. A simple and basic problem-solving technique is exhaustive (also called brute-force) search [154]. Exhaustive search examines all possible solution candidates and chooses the best. Thus, it guarantees an optimal solution if it exists. While it is simple to implement, it is often impractical as the number of solution candidates grows quickly with the problem instance. This effect is known as combinatorial explosion. For most problems, there are many ways to speed up exhaustive search, e.g., skipping invalid, obviously suboptimal, or solutions that are identical to previously examined ones except for permutation. But (automatically) optimizing the pruning of suboptimal solutions may be complex, cumbersome, or impossible. Nevertheless, exhaustive search can be used as part of other (heuristic) approaches.

Backtracking. A common strategy to avoid a full enumeration and prune invalid or suboptimal solutions is backtracking [201]. Backtracking tries to step-by-step extend a partial solution to a full solution. If the partial solution cannot be extended to a valid or optimal solution anymore, the last step or steps are taken back. The algorithm is simple. But the search (or pruning of suboptimal solutions) is very systematic and not fully optimized: For example, early steps are only taken back late. Beginning with different steps to better prune suboptimal solutions may be faster. Overall, the order in which steps are taken is not optimized.

Mathematical Programming. Using mathematical programming, we can formalize the assignment problem with a set of constraints/(in)equations and an optimization function. Then, we can use algorithms that take advantage of the formalized problem structure. These algorithms use a highly systematic enumeration of the search space compared to exhaustive search. As a result, they find optimal solutions relatively quickly. Sedgewick et al. state that even though the search may take exponential time, real-world inputs are evidently not worst-case inputs [201]. As a result, we can speed up the search significantly.

More restricted (e.g., convex or even linear) mathematical formulations enable faster algorithms that can quickly narrow down and find the solution. Many assignment problems are modeled with an (integer) linear programming formulation. Efficient algorithms or paradigms in this field are the simplex algorithm [56], brach-and-bound [55, 130], and interior-point methods [56]. Such algorithms are integrated into mathematical solvers, which can be used as off-the-shelf programs to solve formalized problems. There are commercial (e.g., FICO Xpress [77], Gurobi [94], IBM ILOG CPLEX [109]) and open source (e.g., GLPK [86], lp_solve [18], SCIP [20]) solvers. To speed up the calculation of the actual problem, solvers typically include a presolve phase, in which, for example, redundant constraints are removed, and variables are fixed if possible [33]. Using solvers benefits from algorithmic improvements and hardware improvements [21] because the solvers are steadily optimized with algorithmic advances, and for effective and efficient resource usage. Bixby summarized the history and advances of (integer) linear programming [21, 22]. For example, Bixby reported a combined improvement factor of more than $5 \cdot 10^6 (< 3300 \cdot 1600)$ for the CPLEX LP solver in the sixteen years from 1988 to 2004 due to algorithmic ($3300\times$) and machine ($1600\times$) improvements [22].

Overall, integer linear programming remains an NP-hard problem. The increase in computation time can be mitigated by using highly optimized solvers (and the integrated algorithms), but it may be too high for practical use. Further, the computation time for the optimal solution is (finite but) unknown in advance. Trauth and Woolsey stated that predicting the runtime is generally impossible before trying to solve the problem [218]. Nevertheless, the computation can be aborted to obtain the current best-found solution. Further, it is possible to relax the problem, e.g., dropping the constraint that the variables should have integer values and instead allow fractional values. The solution of the relaxed problem can be obtained faster. Solving the relaxed problem first (in the presolve phase) provides a bounded solution with fractional values [33]. Following, we can also abort the computation when a sufficiently good solution with regard to the theoretical limit has been found.

In this thesis, we use mathematical programming, in particular integer linear programming, to model and solve different data allocation and workload distribution problems. ILP is commonly used in industry and research for various assignment problems, also in the context of physical database optimization, e.g., index tuning [57], data tiering [28, 225], and data encoding selection [26]. The application of ILP is typically limited to obtaining optimal solutions for smaller problem instances. Otherwise, simple greedy heuristics, which are considerably quicker, are often used for larger problems [26, 180, 225]. In contrast, we also use mathematical programming for larger problem instances by (i) exploiting a heuristic problem decomposition, (ii) using solver relaxations, and (iii) clustering of model inputs. Further, we design and evaluate multiple versatile extensions of our integer linear model. For our evaluation, we use one state-of-the-art commercial solver (Gurobi [94]) with a free academic license. A performance (or feature) comparison of different solvers is out of the scope for this thesis.

Although the optimal solution approaches are limited to smaller problem instances (or less complex problem formulations) [205], their solutions are useful to analytically measure any heuristics' performance with regard to the value of the optimization function and calculation time [102, 195]. Further, we can analyze the structure and properties of optimal solutions, which may help to derive heuristic approaches.

4.2.2. Heuristic Approaches

As optimally solving larger assignment problem instances may take practically too long, we require so-called *heuristic* [174] approaches that find (good) solutions faster. There is a broad range of heuristic solving techniques. Some approaches are problem-specific, i.e., they can only be applied to specific optimization problems (e.g., the k-opt method for the traveling salesman problem [142]). But there are also *metaheuristics* [25], which define an abstract order of steps that can be theoretically applied to arbitrary problems. The individual abstract steps of metaheuristics require a problem-specific implementation. Naturally, the transition from problem-specific to more algorithmic-oriented metaheuristics is smooth. Since the late 1990s and the early 2000s, the focus of research on metaheuristics has shifted from an algorithm-oriented to a problem-oriented point of view [24, 206, 213]: Blum et al. stated that the focus is on solving the problem at hand in the best possible way rather than promoting a certain metaheuristic [24]. Talbi summarized that for many optimization problems, the best results are obtained by hybrid algorithms, which combine different algorithmic components [213].

For an overview of some of these algorithmic concepts, we summarize prominent metaheuristics in the following. We focus on explaining the heuristics' core idea rather than covering algorithmic details, variants, or implementation aspects, which are covered in the original papers and many surveys [25, 85, 154, 174, 167]. Although there are also other ways to classify metaheuristics (e.g., based on their origin: nature-inspired vs. non-nature inspired, or based on their usage of search history: memory usage vs. memory-less methods), we primarily divide the algorithms in our overview between *constructive* and *improving* methods. For both classes, we start with *trajectory* methods, which work on a single solution at a time. The name is based on the shape of the investigated solutions over time, which is a trajectory in the search space [25]. They are followed by *population-based* methods, which work on a set of solutions, called the population [25, 85].

Constructive Methods. Constructive methods generate solutions from scratch by adding elements rather than improving complete solutions [25, 85].

- **Basic Greedy Search.** Greedy algorithms make the locally optimal choice in each iteration, i.e., the choice that looks best at the moment [50]. Greedy search can quickly find relatively good solutions, which may then be iteratively refined by (trajectory or population-based) improving methods [25, 85].

Further, other constructive methods are often based on greedy search [85]. However, greedy algorithms are conceptually simple and "may pay for that simplicity by failing to provide good solutions to complex problems with interacting parameters", as Michalewicz and Fogel state [154].

- **Look-Ahead Strategies.** A look-ahead strategy [174] is one extension of the greedy search that evaluates the current choice not only for the current moment but also for several steps into the future. To an extreme, one could evaluate a possible choice by generating a complete solution out of it [67].

- **Ant Colony Optimization.** Ant colony optimization is a population-based method and an example of swarm intelligence, for which the solution is constructed by multiple (partial) solutions of so-called agents. In the case of ant colony optimization, the agents mimic the behavior of ants, which use pheromones to identify good paths during their forage. When an agent has constructed a solution, the "pheromone level" of each solution element is updated to reflect the solution quality. Agents in the following iteration(s) can then use this information to find better solutions.

Improving (Local Search) Methods. Improving methods (try to) iteratively enhance single (trajectory-based) or multiple (population-based) solutions.

- **Basic Hill Climbing.** Based on an initial solution, the algorithm tries to improve the solution incrementally and stops as soon as no further improvement can be found [174]. This process corresponds to a climber that walks only uphill in the fog and assumes to be on the top if he stays on a place where all directions go downhill. Basic hill climbing stops at local maxima (or minima), which may be far from optimal depending on the specific problem. But it is simple and may provide better results than other algorithms when the time is limited (e.g., for interactive applications).

  Getting stuck in local optima is a common limitation for heuristics [81]. To prevent this problem, common strategies use randomness (e.g., simulated annealing) or memory (e.g., tabu search) [81].

- **Simulated Annealing.** The core idea of simulated annealing [116] to overcome local maxima is randomly allowing intermediate solutions with worse quality than the solution before. The probability of choosing such worse solution decreases during the search and is controlled by a "cooling schedule" that mimics the *annealing* process of a crystalline solid [85]. The acceptance of worse solutions also depends on the degradation of the optimization function's value.

- **Tabu Search.** To overcome local maxima (and avoid cycles), tabu search [82, 83] memorizes recently visited solutions and marks them as *tabu* (forbidden).

To overcome local maxima, it also accepts worse solutions if no allowed solution is an improvement. Further, the forbidden solutions prevent cycles, i.e., repeatedly visiting the same intermediate solutions.

- **Evolutionary Computation.** Evolutionary computation algorithms are population-based methods and use computational models of evolutionary processes for problem solving [208]: They iteratively select and then apply a number of operators on solutions of the current population to generate the solutions of the next generation. Solutions are selected based on their fitness values, e.g., the value of the objective function. Operators to produce new solutions include mutations/modifications of single solutions and recombinations (also called crossover) of two or more current solutions [25].

Besides their faster calculation compared to optimal solution approaches, Silver [205] summarizes further potential reasons for utilizing heuristic solution methods. (i) Although not obtaining optimal solutions, people are often happy for improvements over currently achieved results. (ii) People may prefer to understand simpler heuristic approaches to explain their solution over complex routines with unexplainable results and unknown calculation times. In this context, Silver cites Woolsey and Swanson: "People would rather live with a problem they cannot solve than accept a solution they cannot understand" [232]. (iii) Heuristics can be more robust with regard to (slight) data or constraint changes than optimal solutions, which require potentially expensive recalculations [14] (with unpredictable computation times). (iv) Heuristics can be used as part of optimization routines, e.g., to generate initial solutions or to find bounds to quickly identify suboptimal solutions.

Further, the reduced calculation time compared to optimal solutions allows to use more complex models, which may better fit the original problem. In this case, it may be better to achieve a "reasonable (non-optimal) solution to a more accurate model" than an "optimal solution to an incorrect or oversimplified model of the real-world problem" [205].

However, although heuristic approaches often find (good) solutions quickly, heuristics may fail to find a valid solution at all. In this case, one may not be able to determine whether there is no feasible solution or whether the algorithm was unable to find one. Further, it is usually impossible to assess the heuristic solution's quality compared to the optimal value of the objective function.

(Best-)Suitable (meta)heuristics depend on the specific problem formulation and instance. Two key properties of heuristics are *diversification* and *intensification* [25, 84] within the solution space: Diversification is the purposely exploration of uncharted regions of the solution space [84]. Intensification is the more thorough search for solutions in "attractive regions" of the solution space [84]. Naturally, the specific implementation of the metaheuristic (i.e., how well it can be adapted to fit the problem and tuned) may also influence the choice.

4.3. Specific Solution Approaches

After the overview of assignment problems and general solution approaches, this section presents more specific solution approaches. We focus on approaches that have been proposed for the workload distribution in partially replicated database clusters and that we use in our evaluation to compare against (Section 4.3.1). Following, we address specific related problems and approaches in Section 4.3.2.

4.3.1. Workload Distribution for Partially Replicated Database Clusters

In this thesis, we optimize the allocation of queries and fragments to nodes of a database cluster. Our goal is to find a good balance between robust throughput and a low memory consumption. The optimization goal and the constraints are similar to the work of Rabl and Jacobsen [180, 181]. In their work, they present an allocation strategy for database clusters and discuss how to consider data modification costs, node failures, changing workloads, and reallocations. To cope with node failures, Hsiao and DeWitt propose an allocation strategy [108], which we can apply to our problem. In the following, we discuss these allocation approaches and extensions.

Greedy Basic

Rabl and Jacobsen propose a greedy constructive heuristic to generate a suitable allocation [181]. In the following, we first describe their approach. Afterward, we explain it using our exemplary read-only workload (see Figure 4.1 and 4.2).

Initially, no queries or fragments are assigned. The heuristics assigns queries (and the corresponding accessed fragments) to nodes in a query-by-query approach. It starts to assign queries that account for a large workload share and access the most data because these queries potentially cause the highest data duplication if they are assigned late and their load, thus, has to be potentially split across multiple nodes. In specific, the queries are sorted by the product of their workload share (i.e., query frequency f_j $\times$ query costs c_j) and the total size of accessed fragments q_j in descending order. This sequence of queries to assign determines the (current) allocation order. 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. (As a result, all nodes usually get assigned queries and their fragments early.) If a query's workload share would exceed 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. Nodes with an already full load capacity (i.e., $L = 1/K$) are not regarded for the query assignment.

Figure 4.1 and 4.2 visualize the model input and the greedy approach from an initial empty allocation via the intermediate allocations 1 - 6 to the solution, i.e., final allocation. For each allocation state, the sorted queries to assign, allocated fragments

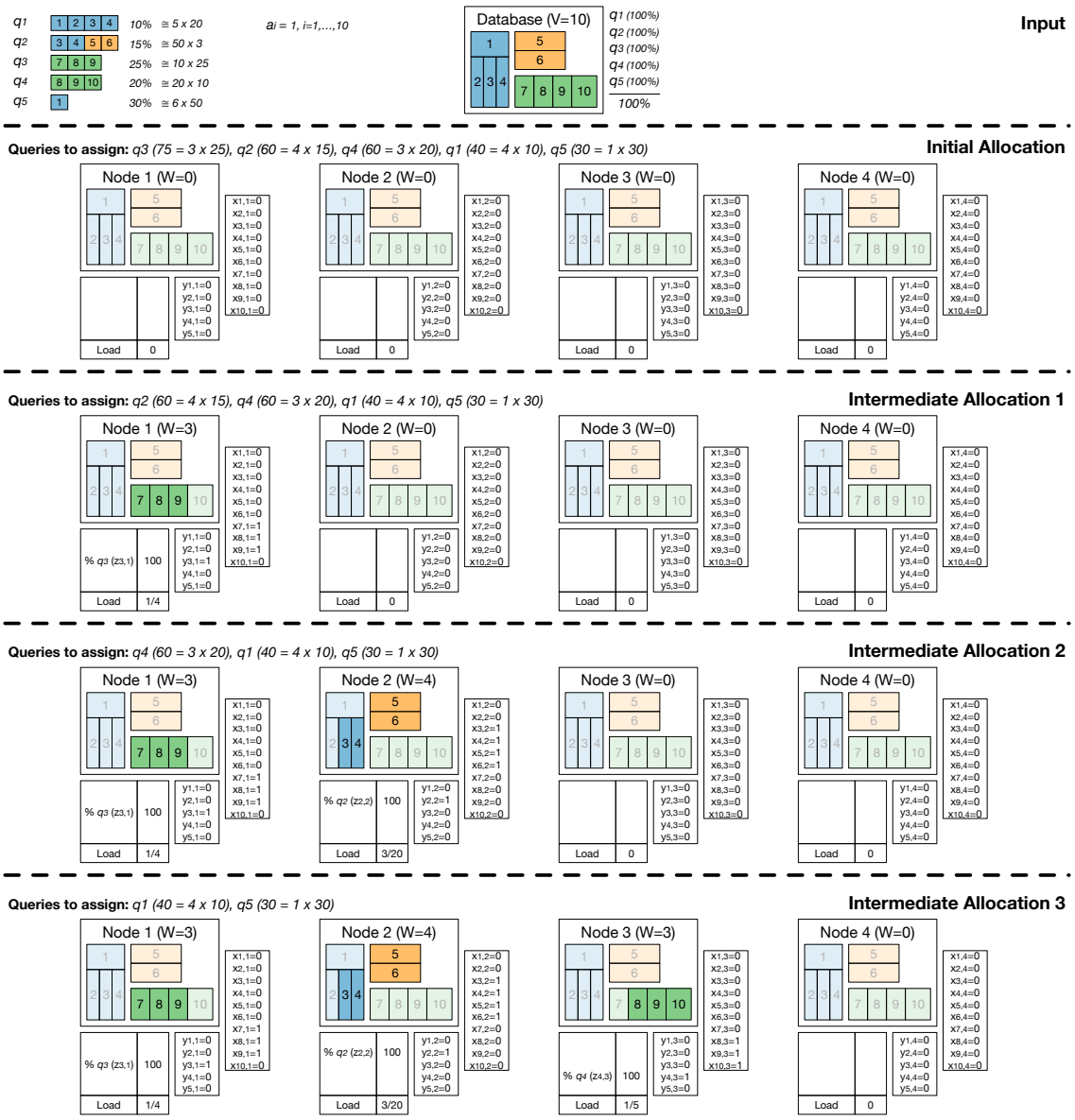


Figure 4.1.: Greedy basic heuristic for the **basic problem**: distributing the workload to $K = 4$ nodes. (1/2)

(colored fragments and $x_{i,k}, i = 1, ..., 10, k = 1, ..., 4$), and assigned workload shares (query load $z_{j,k}$ and executable queries $y_{j,k}$, $j = 1, ...5, k = 1, ..., 4$) are given. In our example, query $j = 3$ (and its three accessed fragments 7 - 9) is assigned first to node $k = 1$ (see intermediate allocation 1). The processing capacity of node $k = 1$ is then full ($L = 1/4$), and the node is not further regarded in the basic algorithm.

Next, query $j = 2$ is assigned to node $k = 2$ (see intermediate allocation 2); then, query $j = 4$ to node $k = 3$ (see intermediate allocation 3); and query $j = 1$ to node $k = 4$ (see intermediate allocation 4). For each assignment, the query load can be

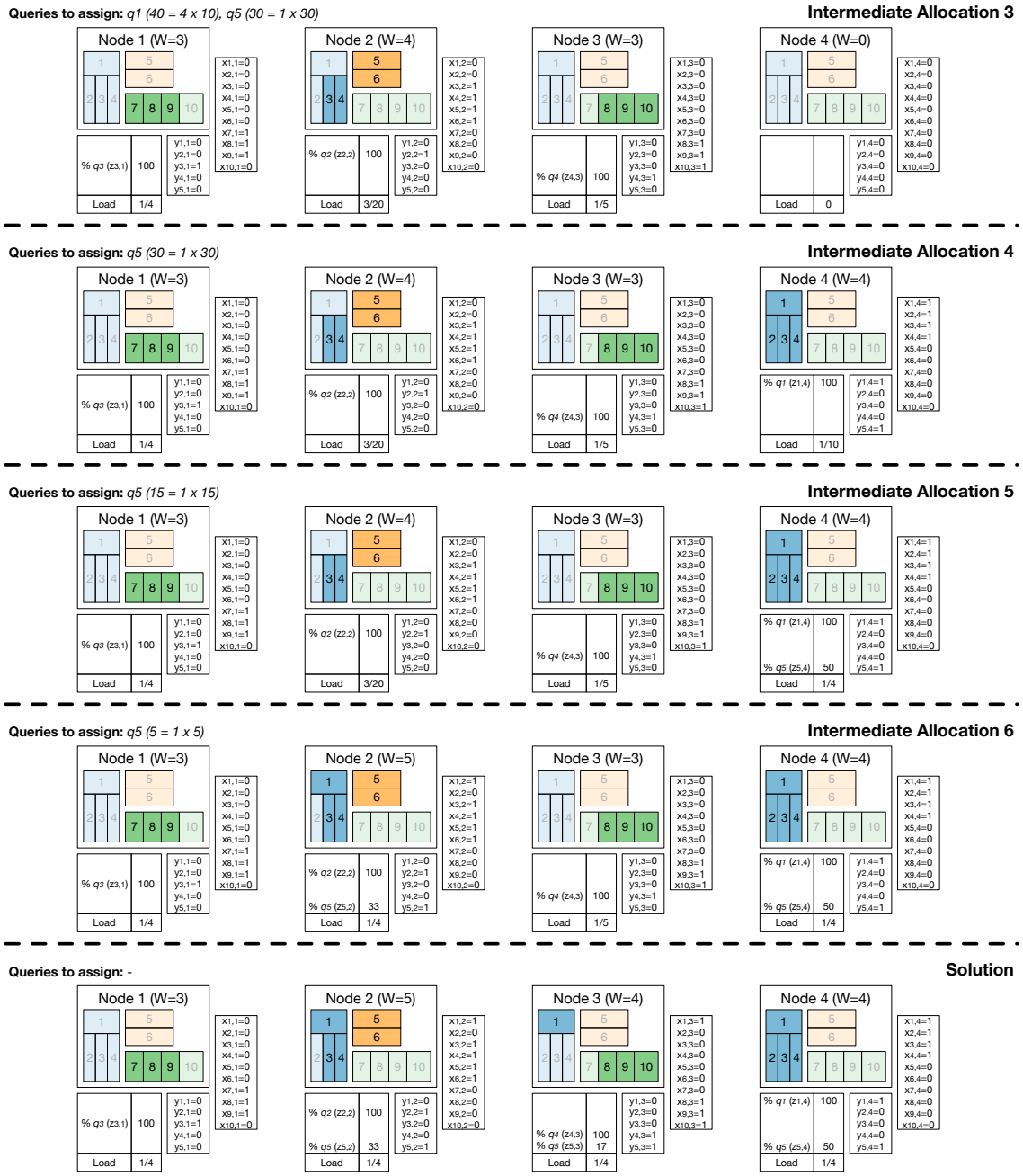


Figure 4.2.: Greedy basic heuristic for the **basic problem**: distributing the workload to $K = 4$ nodes. (2/2)

fully assigned to the selected node. For each assignment, a new node is chosen, as already assigned nodes have no complete fragment overlap.

After the intermediate allocation (state) 4, only query $j = 5$ has to be assigned. The load capacity of node $k = 1$ is full. The other nodes 2 - 4 have remaining load capacities and can be chosen for the assignment. For the next step to assign query

$j = 5$, node $k = 4$ is chosen because it already got assigned fragment $i = 1$ and has, thus, the largest overlap. As assigning the complete load of query $j = 5$ to node $k = 4$ would exceed the load limit of node $k = 4$, the load of query $j = 5$ is split and only assigned partly to node $k = 4$ up to its load limit ($L = 1/4$) (see intermediate allocation 5). The rest load of query $j = 5$ is updated (see intermediate allocation 5) and assigned in the following. Again, it has to be split and is assigned to node $k = 2$ (see intermediate allocation 6) and $k = 3$ (see solution).

As the final result (see Figure 4.2, solution), the workload is entirely assigned and evenly balanced across all nodes (i.e., $L = 1/4$). Overall, $16(= 3 + 5 + 4 + 4)$ fragments are allocated.

The algorithm runs in polynomial time and can quickly calculate allocations for thousands of queries and hundreds of fragments and nodes. We find the heuristic plausible and effective but identified weaknesses due to its simplicity and greedy nature. For example, when ordering the queries, the accessed fragments are not regarded (only their sizes). Further, the remaining queries to assign are not regarded.

Data Modification Costs

Rabl and Jacobsen's base algorithm includes costs for logical data synchronization, i.e., data modifying ("update") queries are assigned to every backend that contains referenced data [181]: For (read-only) queries to assign, the potential costs (and additional fragments) of update queries with overlapping fragments are included. If a read-only query is assigned to a backend, we add (i) the fragments of the related update queries and (ii) the costs of related updates that are not already considered. The overall costs for data modifications depend on the specific assignment of update queries and modified fragments and are unknown in advance. Because the overall modification costs are unknown in advance, the load capacity may have to be adjusted multiple times during the algorithms, which may lead to query assignments to nodes that were previously filled up to their limits.

Greedy Extension for Node Failures

Rabl and Jacobsen propose a greedy approach to complement a basic solution to one that considers node failures [181]: After a basic solution is found, redundant executability of queries is tested and (if necessary) ensured in a query-by-query process, which is similar to the approach of finding a basic solution. 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 considered. As a result, 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.

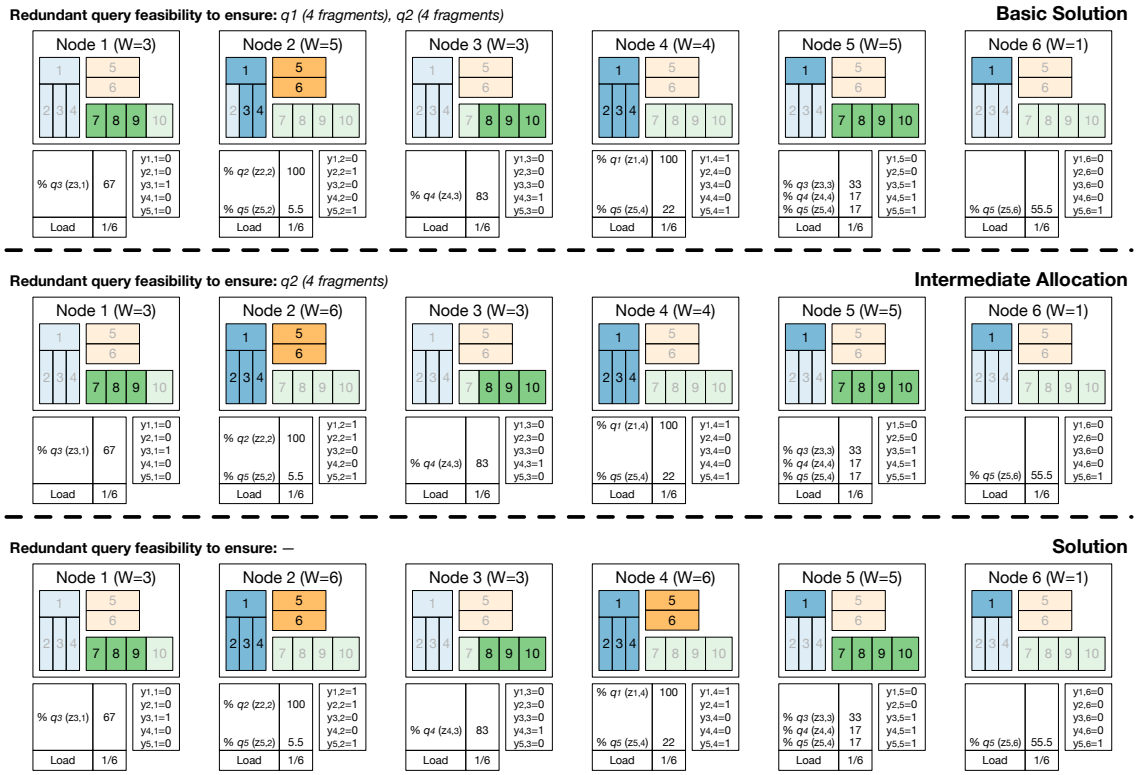


Figure 4.3.: Greedy heuristic extension for **node failures**: distributing the workload to $K = 6$ nodes. Redundant query feasibility is ensured, but if specific nodes fail, single remaining nodes get overloaded.

Figure 4.3 shows this extension for a basic allocation with six nodes. The queries $j = 3, 4, 5$ are already executable by multiple nodes, i.e., the nodes $k = 5, 6$, $k = 3, 5$, and $k = 2, 4, 5, 6$. Query $j = 1$ gets assigned to node 2 because it requires only one additional fragment, i.e., fragment $i = 2$ (see intermediate allocation). Query $j = 2$ gets assigned to node 4, which requires two additional fragments, i.e., fragments $i = 5, 6$ (see solution). To ensure redundant query executability, overall three additional fragments are assigned. Note, no matter which node fails, each query remains executable. However, if, for example, node 2 fails, node 4 becomes overloaded because it is the only node that can process query $j = 1$ and $j = 2$, which account for $10\% + 15\% = 25\%$ of the workload share ($1/5 = 20\%$ of the load are required for an even load balancing).

Chaining Approach for Node Failures

Allocations that guarantee an even workload distribution in failure cases can be constructed by applying the chained declustering strategy to a basic solution [108]: 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

Table 4.1.: Chained declustering scheme for $K = 6$ nodes with the regular load distribution and if node $k = 2$ fails.

Node k (backup of node k)	**1** (6)	**2** (1)	**3** (2)	**4** (3)	**5** (4)	**6** (5)
Regular load $R := 1/K$	1/6	1/6	1/6	1/6	1/6	1/6
Shifted load if $k = 2$ fails	/	- $1R$	- $4/5R$	- $3/5R$	- $2/5R$	- $1/5R$
Taken load if $k = 2$ fails	+ $1/5R$	/	+ $1R$	+ $4/5R$	+ $3/5R$	+ $2/5R$
Load if $k = 2$ fails	1/5	/	1/5	1/5	1/5	1/5

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. This workload-shifting potential enables an even load balancing in single-node failure cases.

Table 4.1 shows an example with $K = 6$ nodes, where node $k = 2$ fails. The regular load is 1/6. If node $k = 2$ fails, its backup node $k = 3$ takes over the load and shifts 4/5 of its regular node to node $k = 4$. Hence, if $k = 2$ fails, the load of node $k = 3$ is $1/6 + 1/6 - (4/5 \cdot 1/6) = 1/5$. Further, node $k = 4$ takes over 4/5 of the load from $k = 3$ and shifts 3/5 of its regular load to node $k = 5$. Resulting, the load of node $k = 4$ is $1/6 + (4/5 \cdot 1/6) - (3/5 \cdot 1/6) = 1/5$. Following this scheme, the load of the nodes $k = 5, 6, 1$ is also 1/5 (see Table 4.1). If node $k = 2$ fails, the workload can be evenly balanced. For other single-node failures, the load shifting works similarly.

Merging Approach for Workload Uncertainty and Reallocations

Rabl and Jacobsen also describe an extension of their approach to cope with multiple workload scenarios [181]: 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. Merged allocations enable an even load balancing for both input allocations. The Hungarian algorithm [127] 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 lost.

The Hungarian algorithm [127] can also be used to calculate the mapping from a current to a new allocation with the smallest reallocation costs. If the current or new allocation should have more nodes, empty nodes (i.e., nodes without fragments) are added, and the Hungarian algorithms can be applied.

4.3.2. Specific Related Problems and Approaches

Archer et al. address an allocation problem that is similar to our basic read-only problem [7]. They evenly load balance queries for web search containing multiple terms, which correspond to the fragments in our model. They use a distributed

balanced graph partitioning tool [11] for clustering queries. By an effective query load balancing, they reduced cache misses by about 50%. In contrast to our model, their system is slightly different. First, data of assigned terms (fragments) can be loaded on demand. Second, the partitioning aims to allocate terms evenly to nodes, whereas our model does not demand an even memory balancing. Nevertheless, the coupled data (term) and query assignment problem is similar to our problem.

In contrast, the allocation problem for replicating data tackled by Ghosh et al. [80] has no such coupling, which significantly reduces the problem's complexity. They replicate fragments according to the access rate and balance the number of fragments per node. Further, they focus on a dynamic setting [151], in which fragments and queries change over time.

There is also a lot of research about data placement in systems with distributed query processing [47, 92] or when allocating data without replication, e.g., when choosing a data storage layout [2, 10, 91] or in the case of multiple storage tiers [28, 132], NUMA nodes [114, 134], or disks [143, 168] (see also Section 2.2.1). Further, some approaches combine data partitioning and replication to reduce the number of distributed transactions [53, 179]. In the context of parallel database systems, Li et al. consider a system with heterogeneous cluster resources [140].

Our work focuses on using partial replication to reduce the overall memory consumption. In contrast, Tashkent+[71] exploits workload-driven load balancing to improve caching, particularly for transaction scaling.

Our workload distribution reduces the memory consumption of replicas. Workload knowledge can also be used to optimize database nodes to process a specific subset of queries more efficiently. We listed these other *divergent design tuning* options (e.g., indices) in Section 2.2 but do not address their specific solution approaches in this thesis.

5

Allocation Models for the Workload Distribution Problem

All models are wrong, but some are useful.

George Box

This chapter presents our optimal and heuristic integer linear programming-based approaches for calculating data fragment allocations for partially replicated database clusters (see Chapter 3). First, we describe solution approaches for the basic problem (Section 5.1), balancing read-only queries. Following, we discuss approaches to flexibly trade computation time for memory efficiency in a targeted way, i.e., how we can sacrifice solution quality to speed up the calculation of allocations if desired (Section 5.2). Finally, we show extensions of our approaches to cover possibly desired allocation aspects (Section 5.3), such as robustness against node failures and uncertain workloads as well as a consideration of data modification and reallocation costs. We shortly summarize our approaches in Section 5.4.

5.1. Solutions for the Basic Problem

In Section 1.1, we identified three allocation goals: a low memory consumption, a short calculation time, and robust performance. In this section, we derive integer linear programming (ILP) models to calculate allocations that enable an even load balancing for read-only queries without optimizing robustness. First, we present a model that minimizes the memory consumption in Section 5.1.1. As optimal solutions are only tractable for small problem instances, we derive a decomposition heuristic based on the optimal model to lower the calculation time in Section 5.1.2.

5.1.1. Optimal Solution

In this section, we describe a model to derive optimal allocations. Table 5.1 summarizes the input parameters, decision variables, and solution variables (see also Chapter 3 for detailed descriptions).

Table 5.1.: Notation table for the optimal model and its adaption.

symbol	description
N	number of fragments $i, i = 1, ..., N$
Q	number of queries $j, j = 1, ..., Q$
K	number of nodes $k, k = 1, ..., K$
a_i	size of fragment $i, i = 1, ..., N$
q_j	accessed fragments of query j, $j = 1, ..., Q$, a subset of {1,...,N}
f_j	frequency of query $j, j = 1, ..., Q$
c_j	costs of processing query $j, j = 1, ..., Q$
C	total workload costs
V	overall size of all accessed fragments
L	a node's workload limit (if no penalty approach is used)
ϵ	admissible deviation of an optimal workload limit
α	penalty factor for the workload limit
β	incentive factor for executable queries
$x_{i,k}$	fragment i is allocated to node k: yes (1) / no (0)
$y_{j,k}$	query j can run on node k: yes (1) / no (0)
$z_{j,k}$	query j's workload share assigned to node k: $\in [0, 1]$
W	overall size of all allocated fragments (memory consumption)
W^*	memory consumption of the optimal solution
W/V	an allocation's normalized memory consumption (replication factor)
L	a node's workload limit (if a penalty approach is used)

We seek to minimize the overall data redundancy W, which is the sum of all stored fragment sizes over all nodes (i.e., $\sum_{i=1,...,N,k=1,...,K} x_{i,k} \cdot a_i$), such that all nodes do not exceed a certain workload limit L, $0 \leq L \leq 1$. To balance the load evenly among all cluster nodes $k, k = 1, ..., K$, we set $L = 1/K$.

Like Rabl and Jacobsen [181], we assume that the query shares z can be chosen without regard to query frequencies and costs, which may be discrete. The decision variables x, y, and z have to be chosen such that the objective

$$\begin{gathered} minimize \\ x_{i,k}, y_{j,k} \in \{0, 1\}, z_{j,k} \in [0, 1], \\ i = 1, ..., N, j = 1, ..., Q, k = 1, ..., K \end{gathered}$$

$$\sum_{i=1,...,N,k=1,...,K} x_{i,k} \cdot a_i \tag{5.1}$$

is minimized and the following (families of) constraints are satisfied:

$$y_{j,k} \cdot |q_j| \leq \sum_{i \in q_j} x_{i,k}, \quad j = 1, ..., Q, k = 1, ..., K \tag{5.2}$$

$$z_{j,k} \leq y_{j,k}, \quad j = 1, ..., Q, k = 1, ..., K \tag{5.3}$$

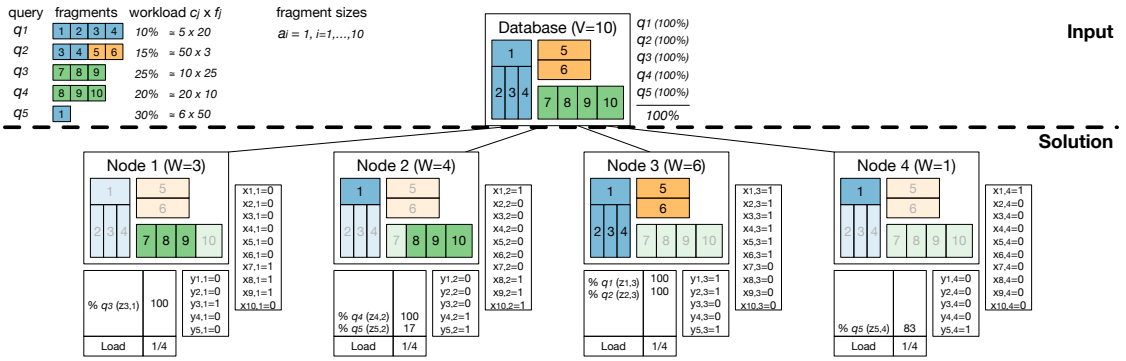


Figure 5.1.: Optimal solution of the **basic problem**: distributing the workload to $K = 4$ nodes.

$$\sum\nolimits_{j=1,...,Q} f_j \cdot c_j / C \cdot z_{j,k} \leq L, \quad k = 1, ..., K \tag{5.4}$$

$$\sum\nolimits_{k=1,...,K} z_{j,k} = 1, \quad j = 1, ..., Q \tag{5.5}$$

The constraints (5.2) guarantee that a query j can only be executed at node k if node k stores all accessed fragments. If we want to execute query j at node k (i.e., $y_{j,k} = 1$), all accessed fragments of query j must be allocated to node k. The cardinality term $|q_j|$ expresses the number of fragments used in query j. The constraints (5.3) ensure that a query j can only have a positive workload share on node k if it can be executed at 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, the constraints (5.3) couple the binary variables y and the continuous variables z in a linear way. Rabl and Jacobsen express this coupling in a nonlinear way using an `if` *condition*, cf. equation (40) in [181]. The constraints (5.4) ensure that no node k exceeds the workload limit $L = 1/K$. The constraints (5.5) guarantee that a query's workload shares over all nodes k sum up to one.

In our basic model, the objective and all constraints are linear in the decision variables. The total number of variables ($N \cdot K + Q \cdot K$ binary and $Q \cdot K$ continuous) and the total number of constraints ($2 \cdot Q \cdot K + K + Q$) increase in the number of fragments N, queries Q, and nodes K. Problem (5.1) - (5.5) is a mixed integer linear programming problem and can be solved using off-the-shelf solvers (see Section 4.2.1).

Figure 5.1 visualizes an exemplary model input with $Q = 5$ queries and $N = 10$ fragments, and an optimal solution for $K = 4$ nodes. The optimal replication factor W/V for an allocation with an even load balancing (i.e., $L = 1/4$) is $(3 + 4 + 6 + 1)/10 = 1.4$. The workload share for all nodes $k, k = 1, ..., 4$, is $L = 1/4$, e.g., for node $k = 3$, $\sum_{j=1,...,5} z_{j,3} = 0.1 + 0.15 + 0 + 0 + 0 = 1/4$. For all queries $j, j = 1, ..., 5$, the sum of query j's shares over all nodes $k, k = 1, ..., 4$, is one, e.g., for query $j = 5$, $\sum_{k=1,...,4} z_{5,k} = 0 + 0.17 + 0 + 0.83 = 1$.

Model Adaptions

The constraints (5.2) and (5.3) (and the objective (5.1)) do not force that $y_{j,k} = 1$ if all fragments of query j are stored on node k but $z_{j,k} = 0$. To enforce $y_{j,k} = 1$ in such cases (to better interpret the values of y), we could adapt the objective as follows:

$$\sum\nolimits_{i=1,...,N,k=1,...,K} x_{i,k} \cdot a_i - \sum\nolimits_{j=1,...,Q,k=1,...,K} y_{i,k} \cdot \beta$$

Using this objective with a small enough $\beta < a_i, i = 1, ...N$, forces the solver to set $y_{j,k} = 1$ if it is possible without allocating additional fragments.

Solvers use floating point arithmetic, which has a finite precision [218]. Due to possible slightly imprecise calculations, it may be impossible for solvers to find a floating point solution with a perfect/even load limit $L = 1/K$. To circumvent this practical obstacle, we can configure the solver to allow minor inaccuracies [78] or adapt our programming model as follows:

- **Load Relaxation.** We can allow a minor (practically insignificant) load imbalance among the nodes and adapt constraints (5.4) with a sufficiently small $\epsilon > 0$ (e.g., $\epsilon = 0.001$) as follows:

  $$\sum\nolimits_{j=1,...,Q} f_j \cdot c_j/C \cdot z_{j,k} \leq (1 + \epsilon) \cdot 1/K, \quad k = 1, ..., K$$

  The deviation of L from a perfect/even load balancing with $L = 1/K$ can be controlled by choosing ε such that a given maximum relative deviation is not exceeded. An absolute load deviation could also be controlled via $1/K + \epsilon$. Note, load relaxations with an increasing ϵ make it possible to obtain a memory consumption W that is smaller than W^*, which is optimal for $L = 1/K$ (i.e., $\epsilon = 0$).

- **Penalty Approach.** We can use a common penalty approach and adapt objective (5.1): Instead of using a fixed parameter L, the workload limit is a variable and penalized with a sufficiently large penalty factor $\alpha > 0$ (e.g., $\alpha = 1000$) in the objective while the data redundancy is normalized by the overall size of all accessed fragments V as follows:

  $$\begin{gathered} minimize \\ x_{i,k}, y_{j,k} \in \{0, 1\}, z_{j,k} \in [0, 1], 0 \leq L, \\ i = 1, ..., N, j = 1, ..., Q, k = 1, ..., K \end{gathered}$$

  $$1/V \cdot \sum\nolimits_{i=1,...,N,k=1,...,K} x_{i,k} \cdot a_i + \alpha \cdot L$$

  By choosing α, we can weigh the tradeoff between a low memory consumption (with a small α) and an even load balancing (with a large α). If the

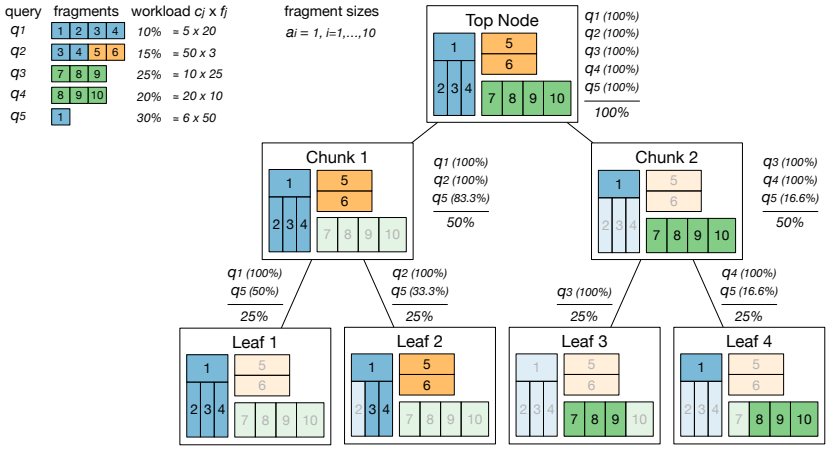


Figure 5.2.: Decomposition-based heuristic solution of the **basic problem**: decomposing the workload to $K = 2 + 2 = 4$ final nodes.

penalty factor α is sufficiently large, the solution guarantees the smallest possible workload limit $L^* = 1/K$. Similarly to increasing ϵ in the load relaxation approach, a decreasing α makes it possible to achieve $W < W^*$ while the load balancing becomes suboptimal $L > L^* = 1/K$.

Using the workload limit L as a variable and fixed fragment allocations $x_{i,k}, i = 1, ..., N, k = 1, ..., K$, as parameters, we can calculate the worst (highest) workload share over all nodes for a given workload scenario. Note, fixed fragment allocations $x_{i,k}, i = 1, ..., N, k = 1, ..., K$, determine the executable queries per node $y_{j,k}$ (see constraints (5.2)). When solving the ILP, the remaining variables $z_{j,k}$ and L are chosen such that L is as small as possible and, thus, coincides with the worst-case workload share over all nodes. If $L = 1/K$, the allocation enables an even load balancing for the workload.

5.1.2. Decomposition-Based Heuristic

The complexity of our optimal ILP model (5.1) - (5.5) grows with the number of fragments N, queries Q, and nodes K. As a result, calculation times can get too large (see Section 3.2.1) when considering huge workloads with hundreds of query classes, fragments, and nodes. Effective allocations assign queries that access the same fragments to the same node. We do not need to assign queries to nodes in a single step. Instead, we propose a decomposition-based heuristic, which assigns queries to nodes recursively, using *multiple easier-to-solve generalized ILP subproblems*. The subproblems form a tree with K leaves: The root node corresponds to the overall workload. The leaves correspond to the K nodes of the optimal model. Intermediate nodes correspond to chunks of queries that access similar fragments. Figure 5.2 illustrates a decomposition of our example workload with $K = 4$ final nodes.

Table 5.2.: Notation table for the decomposition-based heuristic.

symbol	description
N	number of fragments $i, i = 1, ..., N$
Q	number of queries $j, j = 1, ..., Q$
K	(overall) number of nodes $k, k = 1, ..., K$
$\boldsymbol{B}$	number of subnodes $b, b = 1, ...B$
$\boldsymbol{\bar{x}_i}$	fragment i is relevant: yes (1) / no (0)
$\boldsymbol{\bar{y}_j}$	query j is relevant: yes (1) / no (0)
$\boldsymbol{\bar{z}_j}$	query j's workload share: $\in (0, 1]$
a_i	size of fragment $i, i = 1, ..., N$
q_j	accessed fragments of query j, $j = 1, ..., Q$, a subset of {1,...,N}
f_j	frequency of query $j, j = 1, ..., Q$
c_j	costs of processing query $j, j = 1, ..., Q$
C	total workload costs
V	overall size of all accessed fragments
L	the load limit of leaf nodes
$\boldsymbol{n_b}$	the number of represented leaf nodes per subnodes $b, b = 1, ..., B$
$\boldsymbol{L_b}$	the load limit of individual subnodes $b, b = 1, ..., B$
$x_{i,b}$	fragment i is allocated to subnode b: yes (1) / no (0)
$y_{j,b}$	query j can run on subnode b: yes (1) / no (0)
$z_{j,b}$	query j's workload share assigned to node k: $\in [0, 1]$
$\boldsymbol{W^D}$	memory consumption of the decomposition-based ILP heuristic

We generalize our ILP model (5.1) - (5.5) so that it can be applied in any node of the tree to split the node's corresponding workload into chunks of similar queries: Consider an arbitrary (workload) node in the tree. The node's workload is given by the relevant fragments $\bar{x}_i := 1$, relevant queries $\bar{y}_j := 1$, and workload shares $\bar{z}_j \in (0, 1]$, $i \in I \subseteq \{1, ..., N\}$, $j \in J \subseteq \{1, ..., Q\}$. We want to split the workload to a number of subnodes $B, 1 \leq B \leq K$. Each subnode b represents a number of leaves (final nodes) n_b, $b = 1, ..., B$. Hence, a subnode b has to take the workload share $L_b := n_b/K$, $b = 1, ..., B$. Table 5.2 summarizes the input parameters, decision variables, and solution variables for our decomposition heuristic. New parameters and variables are highlighted.

Next, we present our generalized ILP model to flexibly decompose the workload:

$$
\begin{aligned}
&\qquad\qquad minimize \\
&x_{i,b}, y_{j,b} \in \{0, 1\}, z_{j,b} \in [0, 1], \\
&i = 1, ..., N, j = 1, ..., Q, b = 1, ..., B : \bar{x}_i = 1, \bar{y}_j = 1
\end{aligned}
$$

$$
\sum\nolimits_{i=1,...,N, b=1,...,B:\bar{x}_i=1} x_{i,b} \cdot a_i \tag{5.6}
$$

subject to:

$$y_{j,b} \cdot |q_j| \leq \sum\nolimits_{i \in q_j} x_{i,b}, \quad \begin{matrix} j = 1, ..., Q : \bar{y}_j = 1 \\ b = 1, ..., B \end{matrix} \tag{5.7}$$

$$z_{j,b} \leq y_{j,b}, \quad \begin{matrix} j = 1, ..., Q : \bar{y}_j = 1 \\ b = 1, ..., B \end{matrix} \tag{5.8}$$

$$\sum_{j=1,...,Q:\bar{y}_j=1} f_j \cdot c_j / C \cdot z_{j,b} \leq L_b, \quad b = 1, ..., B \tag{5.9}$$

$$\sum\nolimits_{b=1,...,B} z_{j,b} = \bar{z}_j, \quad j = 1, ..., Q : \bar{y}_j = 1 \tag{5.10}$$

The objective (5.6) minimizes the memory consumption over all subnodes and corresponds to the objective (5.1) of the basic model. The constraints (5.7) and (5.8) correspond to the constraints (5.2) and (5.3) of the basic model but consider only relevant queries $j = 1, ..., Q : \bar{y}_j := 1$. The constraints (5.9) correspond to the constraints (5.4) of the basic model but use individual load shares per subnode L_b, depending on the number of represented leaves $n_b, b = 1, ..., B$. The constraints (5.10) correspond to the constraints (5.5) of the basic model but use individual workload shares per query $\bar{z}_j, j = 1, ..., Q : \bar{y}_j := 1$.

Overall, the ILP model (5.6) - (5.10) can split workload subsets flexibly, i.e., it is a generalization of the optimal model (5.1) - (5.5), for which $B := K$, $L_b := 1/K, b = 1, ..., B$, and $\bar{x}_i := 1$, $\bar{y}_j := 1$, $\bar{z}_j := 1$, $i = 1, ..., N$, $j = 1, ..., Q$.

When decomposing a workload using multiple ILP subproblems, we characterize the total workload by $\bar{x}_i := 1$, $\bar{y}_j := 1$, $\bar{z}_j := 1$, for all $i = 1, ..., N$ and $j = 1, ..., Q$. Let the optimal solution of an ILP model (5.6) - (5.10) be denoted by x^*, y^*, and z^*. Then, the remaining workload of each subnode b is characterized by $\bar{x}_{\bar{i}} := 1$, $\bar{i} \in \{i = 1, ..., N : x^*_{i,b} = 1\}$, $\bar{y}_{\bar{j}} := 1$, and $\bar{z}_{\bar{j}} := z^*_{j,b}$, $\bar{j} \in \{j = 1, ..., Q : y^*_{j,b} = 1\}$, $b = 1, ..., B$. For each subnode b, the model (5.6) - (5.10) can be applied again. From level to level, the number of relevant fragments and queries decreases. Thus, the number of variables and constraints gets smaller (for constant B). On each level and for each node, the number of subnodes B and their represented leaves n_b, $b = 1, ..., B$, can be chosen arbitrarily. This way, all integer numbers K can be decomposed, and the final workload distribution guarantees an even load balancing, i.e., $L = 1/K$.

The number of subnodes B can be used to *control* the problem complexity. The smaller B, the faster is the computation (on each level), but the overall memory consumption might increase (compared to optimal allocations). To obtain minimal computation times, starting with $B = 2$ on the highest level is advantageous.

The problem complexity gets quickly smaller with each decomposition from the root to the leaves. However, if the number of fragments and queries are large, the ILP approach might take too long, even if the number of chunks is small. In such cases, we could decompose large subproblems (at a tree's root) heuristically,

particularly using Rabl and Jacobsen's greedy heuristic [181]; if subproblems are sufficiently small, the ILP approach can be used henceforth (towards a tree's leaves). We can also use additional techniques to lower the calculation time for ILP-based approaches, which we explain in the following.

5.2. Approaches to Lower the Computation Time

Compared to the optimal solution, our decomposition approach lowers the problem complexity by reducing the number of nodes, and the number of queries and fragments in the lower levels. We can adjust the decomposition's chunking (i.e., the number of decomposition levels and chunk sizes) to control the complexity of the ILP models and, thus, the calculation time. Nevertheless, for large problem inputs, the calculation time of our decomposition approach may still be too high to be applicable in practice – even if we use the smallest chunk sizes with many levels. The high calculation times are because solvers *guarantee* (and thus have to find) *optimal solutions* while the *solution space* of valid allocations *is huge*.

In this section, we discuss techniques to further lower the calculation time of ILP-based allocation approaches. First, we explain how we can control the solver's calculation time in Section 5.2.1. Then, we present possibilities to restrict the solution space in Section 5.2.2.

5.2.1. Solver-Based Relaxation Techniques

ILP models can become complex for large-scale workloads: Due to the high number of variables and constraints, the computation time of solvers becomes too large to be applicable in practice. To speed up the computation, we can use relaxed optimality gaps (i.e., the solver runs until a certain *relaxed* optimality goal can be guaranteed) or limit the solver's calculation time. Naturally, both approaches negatively affect the solution quality. However, practitioners are often willing to sacrifice a small amount of the optimal performance to obtain a fast heuristic.

Optimality Gap

One possibility to lower the solver's computation time is to use an optimality gap. Using the objective function, we can calculate a solution's objective value, e.g., the overall memory consumption. Solvers terminate when the objective value cannot be further improved, i.e., when the optimal solution is obtained. Thereby, optimality can be controlled with solver settings: We can set the maximum allowed *relative gap* and *absolute gap* between the desired solution and an upper (theoretical) bound. The solver, then, runs until a certain *relaxed* optimality goal can be guaranteed, i.e., if the *ratio* or *difference* of the current objective value of the ILP problem compared to a theoretical upper bound (e.g., derived by continuous relaxations) is sufficiently small.

The solver Gurobi [94] (version 9.5.2) has a default value of 10^{-4} for the relative gap and 10^{-10} for the absolute gap. A relative optimality gap of 0.1 (10%) guarantees a 90 - 100% solution (with regard to the objective), e.g., having an up to 10% larger memory consumption than optimal. In return, the runtime is shorter (compared to a solution with an optimality gap of zero) but still unknown in advance. Naturally, a larger optimality gap leads to a smaller runtime.

In order to be able to interpret the optimality gap in terms of the memory consumption W/V, we avoid using the penalty approach (see Section 5.1.1), whose load term $\alpha \cdot L$ influences the objective. We could still use the load relaxation (see Section 5.1.1) if required.

Time Limit

Another possibility to speed up the computation is to terminate the solver before the optimal solution is obtained, i.e., we set an upper bound (i.e., limit) for the computation time. The solver runs until the optimal solution is obtained or the given time limit is exceeded. In contrast to using optimality gaps, we can now control the runtime, but the final optimality gap (i.e., solution quality) is not known.

The suitability of optimality gaps and time limits depends on whether solutions must be computed quickly (e.g., for frequent adaptions or online settings) or whether the quality of solutions is in focus (e.g., for offline or overnight computations). The usage of an optimality gap and time limit can also be combined to obtain a solution with a guaranteed performance or within a limited computation time. Such relaxed formulations, which still ensure certain performance guarantees, allow for quick and pragmatic solutions and can be highly beneficial, especially for practitioners.

5.2.2. Techniques to Reduce the Problem Size

The solution space is bounded by the number of decision variables x, y, and z, which are defined over the number of fragments, queries, and nodes. With the decomposition approach, we can lower the number of nodes via a lower number of intermediate workload chunks.

To further reduce the solution space, we can decrease the number of decision variables by clustering multiple queries (or fragments) to a single query (or fragment) group. A coarser granularity of query classes (and fragments) naturally decreases the optimization potential but also the solution space, which speeds up the calculation.

Further, we can limit the solution space by pinning a subset of fragments (or queries) to nodes so that the corresponding input variables are fixed. We can, for example, use an optimized basic solution as allocation input to derive an extended robust one.

Both approaches are heuristics, which may lower the solution quality but, in return, speed up the calculation (if desired). In the following, we describe our

query clustering in detail. Specific approaches that build on basic or intermediate allocations are covered in the model extensions in the following sections.

Query Clustering

Workload characteristics (e.g., the distribution of query workload shares and accessed fragments) are often skewed [27, 195]. Typically, a small share of the queries represents the majority of the workload costs. Hence, we can directly infer an easy and effective way to cluster queries and, in turn, reduce the complexity of the workload distribution problem:

(i) The majority of queries that represent only a small share of the workload can be clustered within a set denoted by Q^F and *fixedly* assigned to a single (potential full) replica, e.g., the primary node.

(ii) The set of *remaining* (costly) queries denoted by

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

is used as (a smaller) input for the fragment allocation problem with K nodes (including the (full) replica of step (i)).

To determine the set of fixedly assigned queries Q^F, we give two approaches:

- **Full Clustering Approach.** We order the queries j by their workload share $f_j \cdot c_j/C$ in *increasing* order, $j = 1, ..., Q$, i.e., query $j = 1$ has the smallest and query $j = Q$ the highest workload share. Now, we assign *as many queries with small workload shares as possible* to one (e.g., the first) of the K nodes until the accumulated workload share of those queries does not exceed $1/K$, i.e., we set $z_{j,1} = 1$ for all $j \in Q^F$.

  The remaining queries Q^R must be allocated to only $K - 1$ nodes. We must, therefore, adapt the remaining workload and the total workload costs C accordingly in the used allocation model (e.g., (5.1) - (5.5)). Note, the set of fixedly assigned queries depends on the number of nodes K (i.e., $Q^F = Q^F(K)$). If the workload share of the node ($k = 1$) with the fixed queries is (significantly) less than $1/K$, the workload share of node 1 can be filled up with a share of a remaining query.

  Summarizing, the workload distribution problem boils down to a problem with $K - 1$ nodes and a highly reduced complexity (due to fewer queries).

- **Partial Clustering Approach.** We order the queries by their workload share in increasing order, equal to the full clustering approach. However, we assign only the $|Q^F|$ queries with the smallest workload share to one (e.g., the first) of the K nodes, whereby the number $|Q^F|$ is sufficiently small such that the workload share of the associated queries is significantly smaller than $1/K$.

The remaining workload Q^R must be allocated to the other $K-1$ nodes and the *residual resources* of the first cluster node ($k=1$). We can directly include this partial clustering heuristic in our ILP model via the constraints

$$z_{j,1} = 1 \quad \forall j \in Q^F. \tag{5.11}$$

Note, the constraints (5.11) imply $y_{j,1} = 1$ for all $j \in Q^F$ and the allocation of all required fragments to the cluster node 1. If the decomposition approach is used, the constraints (5.11) are only used for chunks that are associated to the leaf node $k = 1$.

Summarizing, one cluster node ($k = 1$) is *not completely* filled up with queries that have small workload shares. Leaving some space on the cluster node allows the solver to assign other data-intensive queries (with a slightly higher workload share) to this node, which can, in turn, reduce the total replication factor. The computation is still sped up significantly if the number of fixed queries $|Q^F|$ is chosen sufficiently large.

Summary

Using integer linear programming-based heuristics, we are flexible to trade calculation time and solution quality in a targeted way. First, we can use solver-based relaxation settings, i.e., optimality gaps and time limits, without modifying the programming model. Further, we can reduce the solution space by clustering queries or fixing fragments to nodes, particularly when workloads are skewed, which is typical for real-world workloads.

For all techniques, the memory consumption of the resulting allocation may be larger than optimal, but the computation time becomes significantly faster. These techniques are especially useful for larger (and skewed) input sizes and more complex problem formulations that use extensions.

We can also combine solver-based relaxations and a query clustering heuristic, and use them on top of our decomposition-based heuristic. Combining these three techniques allows balancing the solution quality and computation time in a targeted way such that optimized solutions can be computed within short and plannable response times.

5.3. Model Extensions

So far, we calculated *basic allocations*, which enable an even load balancing for read-only queries of a single fixed workload scenario, given by the number of nodes and queries. In practice, allocations must consider further aspects (see Section 3.2), such as node failures, workload uncertainty, data modifications, and reallocation costs. This section presents how we can extend our integer linear programming (ILP) models to take these aspects into account.

5.3.1. Node Failures

In the following, we show how we compute allocations that consider node failures. Thereby, a database cluster can support different levels of redundancy.

Redundant Fragment Availability and Query Feasibility:

- **ILP-Based Feasibility Extension.** Allocations may ensure that each fragment is stored at multiple nodes or each query is executable at multiple nodes [181]. We can take a basic ILP-based approach (e.g., our optimal model (5.1) - (5.5)) and add further constraints to ensure that each fragment is stored on multiple (e.g., at least two) nodes, i.e.,

$$\sum_{k=1,...,K} x_{i,k} \geq 2, \quad i = 1, ..., N,$$

  or each query can be executed by multiple (e.g., at least two) nodes, i.e.,

$$\sum_{k=1,...,K} y_{j,k} \geq 2, \quad j = 1, ..., Q.$$

- **Greedy Extension for Node Failures.** For allocations that do not base on ILP, Rabl and Jacobsen [181] propose a greedy approach to complement a basic solution to a more robust one (see Section 4.3.1).

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

Even Load Balancing in Failure Cases:

- **Chaining Approach for Node Failures.** Allocations that guarantee an even workload distribution in failure cases can be obtained by applying the chained declustering [108] strategy to a basic solution: Nodes are chained, forming a ring. Each node acts as backup of its predecessor node (see Section 4.3.1). However, chaining entire nodes may lose optimization potential.

- **Adding Full Replicas.** Another simple heuristic to ensure an even workload distribution in failure cases is to use a basic solution for $K - K^F$ nodes (e.g., $K^F = 1$) and add K^F full replicas. The reason why this works is the following: If the K^F full replicas fail, the $K - K^F$ solution balances the load evenly by definition. Otherwise, the remaining full replicas can take over the workload share of any failed partial node because they can execute all queries. However, in general, using full replicas may be too costly. Hence, sophisticated approaches are needed to guarantee a balanced workload distribution in the case of node failures, particularly for the standard case $K^F = 1$.

Optimal Model

In the following, we present an ILP model to compute robust allocations that simultaneously optimize memory consumption and load balancing in failure cases: We consider the case that one of the nodes $k = 1, ..., K$ might fail. No matter which node is affected, the optimal workload distribution without the failed node should guarantee not only redundant query feasibility but also an even load balancing among the $K - 1$ remaining nodes.

To express the even load balancing in failure scenarios, we have to extend our model: Recap, by L, we denote the highest workload share of all nodes in the basic model without a failure. The limit L is determined by the allocation $x_{i,k}$ and the assigned workload shares $z_{j,k}$, $i = 1, ..., N$, $j = 1, ..., Q$, $k = 1, ..., K$.

Similar to the *regular workload limit* L, we introduce the *emergency workload limit* $L^{(-)}$ for the highest (worst-case) workload share over all $K - 1$ remaining nodes in the case of a potential node failure. To determine and optimize the limit $L^{(-)}$, we introduce the additional variables $\tilde{z}_{j,k^{(-)},k^{(+)}} \in [0, 1]$, which describe the workload share of query j on a remaining node $k^{(+)} \in \{1, ..., K\}\backslash\{k^{(-)}\}$ in case node $k^{(-)} = 1, ..., K$ fails. Table 5.3 summarizes the input parameters, decision variables, and solution variables for our (optimal and heuristic) ILP models that consider node failures. New model parameters and variables are highlighted.

Next, we present our ILP formulation that allows for optimal robust solutions in the case of single-node failures:

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

$$1/V \cdot \sum_{i=1,...,N,k=1,...,K} a_i \cdot x_{i,k} + \alpha \cdot L^{(-)} + \alpha/100 \cdot L \tag{5.12}$$

subject to constraints for the *regular* scenario:

$$y_{j,k} \cdot |q_j| \leq \sum_{i \in q_j} x_{i,k}, \quad j = 1, ..., Q, k = 1, ..., K \tag{5.13}$$

$$z_{j,k} \leq y_{j,k}, \quad j = 1, ..., Q, k = 1, ..., K \tag{5.14}$$

$$\sum_{j=1,...,Q} f_j \cdot c_j / C \cdot z_{j,k} \leq L, \quad k = 1, ..., K \tag{5.15}$$

$$\sum_{k=1,...,K} z_{j,k} = 1, \quad j = 1, ..., Q \tag{5.16}$$

and constraints for the *failure* scenarios:

$$\tilde{z}_{j,k^{(-)},k^{(+)}} \leq y_{j,k^{(+)}}, \quad \begin{array}{l} j = 1, ..., Q \\ k^{(-)} = 1, ..., K \\ k^{(+)} \in \{1, ..., K\}\backslash\{k^{(-)}\} \end{array} \tag{5.17}$$

Table 5.3.: Notation table for ILP models that consider node failures.

symbol	description
N	number of fragments $i, i = 1, ..., N$
Q	number of queries $j, j = 1, ..., Q$
K	number of nodes $k, k = 1, ..., K$
a_i	size of fragment $i, i = 1, ..., N$
q_j	accessed fragments of query j, $j = 1, ..., Q$, a subset of {1,...,N}
f_j	frequency of query $j, j = 1, ..., Q$
c_j	costs of processing query $j, j = 1, ..., Q$
C	total workload costs
V	overall size of all accessed fragments
α	penalty factor for the workload limit
$\boldsymbol{x_{i,k}^{(f)}}$	fragment i is fixedly allocated to node k: yes (1) / no (0)
$\boldsymbol{E}$	(optional) normalized memory budget for added data
$x_{i,k}$	fragment i is allocated to node k: yes (1) / no (0)
$y_{j,k}$	query j can run on node k: yes (1) / no (0)
$z_{j,k}$	query j's workload share assigned to node k: $\in [0, 1]$
$\boldsymbol{\tilde{z}_{j,k^{(-)},k^{(+)}}}$	query j's workload share assigned to a remaining working node $k^{(+)}$ if node $k^{(-)}$ fails: $\in [0, 1]$, $k^{(-)} = 1, ..., K$, $k^{(+)} \in \{1, ..., K\} \backslash \{k^{(-)}\}$
$\boldsymbol{x_{i,k}^{(e)}}$	fragment i is added to node k: yes (1) / no (0)
$\boldsymbol{W^{F*}}$	memory consumption of the optimal solution considering node failures
$\boldsymbol{W^{F}}$	memory consumption of the ILP heuristic considering node failures
L	a node's workload limit
$\boldsymbol{L^{(-)}}$	workload limit in case of a node failure

$$\sum_{j=1,...,Q} \frac{f_j \cdot c_j}{C} \tilde{z}_{j,k^{(-)},k^{(+)}} \leq L^{(-)}, \; \begin{matrix} k^{(-)}, k^{(+)} = 1, ..., K \\ k^{(+)} \neq k^{(-)} \end{matrix} \tag{5.18}$$

$$\sum_{k^{(+)}=\{1,...,K\}\backslash\{k^{(-)}\}} \tilde{z}_{j,k^{(-)},k^{(+)}} = 1, \quad \begin{matrix} j = 1, ..., Q \\ k^{(-)} = 1, ..., K. \end{matrix} \tag{5.19}$$

The objective (5.12) minimizes the replication factor W/V and uses a penalty approach (see Section 5.1.1) for the largest workload shares L (regular case) and $L^{(-)}$ (failure case). To emphasize the failure scenarios, the penalty on $L^{(-)}$ is chosen much larger (e.g., $\times 100$) than the penalty for L. The use of two penalty terms in the objective allows to simultaneously account for *both* workload limits: the regular one (denoted by L) and the emergency one (denoted by $L^{(-)}$).

The constraints (5.13) - (5.16) correspond to the constraints (5.2) - (5.5) of the basic model.

The constraints (5.17) - (5.19) ensure admissible allocations in case a node $k^{(-)} = 1, ..., K$ is not working. Recap, the additional variables $\tilde{z}_{j,k^{(-)},k^{(+)}} \in [0, 1]$ express the

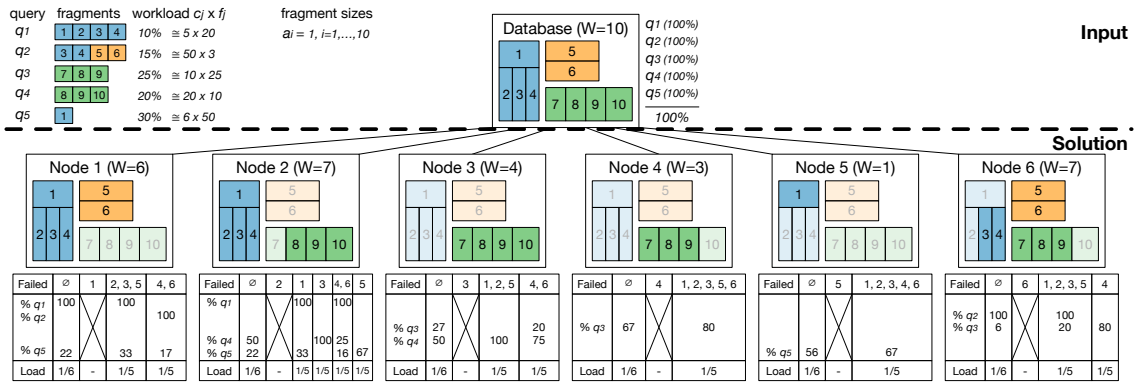


Figure 5.3.: Optimal solution that considers **node failures**: allowing for changed workload distributions for different scenarios; illustration for $K = 6$ nodes, a regular workload limit $L^* = 1/6$ and a guaranteed worst-case workload limit (denoted by $L_{\max}^{(-)*}$) of $1/5$.

optimal workload share of query j on the remaining nodes $k^{(+)} = \{1, ..., K\} \backslash \{k^{(-)}\}$ if node $k^{(-)}$ does not work. For a failure of a fixed node $k^{(-)}$, the constraints (5.17) - (5.19) correspond to the constraints (5.13) - (5.16); the difference is that the constraints (5.17) - (5.19) consider $K-1$ remaining nodes while the constraints (5.13) - (5.16) consider all K nodes. Thus, the ILP model (5.12) - (5.19) *simultaneously* captures the scenario without a failure and the K scenarios in which one node fails.

If the penalty factor α is sufficiently large, the optimal solution for node failures guarantees the smallest possible emergency workload limit $L^{(-)*} = 1/(K-1)$, $K > 1$.

Figure 5.3 shows an optimal solution that considers single-node failures with $K = 6$ nodes for our exemplary workload. While minimizing the overall memory consumption, the solution enables an even load balancing without any node failure $L = 1/6$ and for each of the $K = 6$ single-node failure cases $L^{(-)} = 1/5$. In the tables of Figure 5.3, the values $z_{j,k}$ are shown in the columns ∅ (no node failure). For example, if all nodes are running, the workload share of query $j = 4$ is split among the nodes $k = 2$ ($z_{4,2} = 50\%$) and $k = 3$ ($z_{4,3} = 50\%$). The columns 1 - 6 (node $k^{(-)} = 1, ..., 6$ failed) contain the values $\tilde{z}_{j,k^{(-)},k^{(+)}}$. For example, if node $k^{(-)} = 6$ fails, the workload share of query $j = 4$ is split among the nodes $k = 2$ ($\tilde{z}_{4,6,2} = 25\%$) and $k = 3$ ($\tilde{z}_{4,6,3} = 75\%$).

The fact that even solutions of small problems, as illustrated in Figure 5.3, cannot be easily derived manually indicates the complexity of the problem: To identify an allocation that allows for different workload distributions and at the same time has low data redundancy is highly challenging because K potential failures cases and the non-failure case must be considered at the same time.

Decomposition-Based Approaches

For also calculating robust allocations that enable an even load balancing in failure cases for larger problems, we propose a decomposition-based approach: Instead of solving the problem in a single step (i.e., with a single ILP model), we use multiple steps, which correspond to simpler ILP models, to decrease the overall calculation time. In general, we can *flexibly* combine (i) our scalable decomposition-based heuristic (see Section 5.1.2), (ii) our optimal model for node failures (5.12) - (5.19), and (iii) optimized fragment enhancements. The specific combination of steps can be adjusted to trade memory consumption, computation time, and the emergency workload limit. Next, we first explain our *three-step approach* [104], which combines a low memory consumption and calculation time while enabling an even load balancing for single-node failures. Following, we present a derived *two-step approach* resulting in a lower memory consumption but higher calculation time.

Three-Step Approach. After a short overview of the three steps, we explain the individual steps in more detail. Figure 5.4 illustrates the three-step approach using the same model inputs as in Figure 5.3. Step 1 *consistently splits the workload into chunks of similar queries*, accessing the same fragments. In this step, we use our decomposition-based heuristic (see Section 5.1.2). In step 2, we *compute robust fragment allocations for individual chunks of smaller sizes* such that a node failure is compensated by the remaining nodes of the affected chunk. In this step, we use the model for optimal solutions in failure cases (5.12) - (5.19). Based on the allocations derived in step 2, in step 3, we *compute optimal fragment enhancements* to obtain a perfect load balance *over all nodes*.

- **Step 1: Initial Basic Decomposition.** We assume a given workload characterized by queries and fragments (the upper part of Figure 5.4). In step 1, we split the workload iteratively using our basic decomposition-based heuristic (see Section 5.1.2), forming a tree of chunks with similar queries, which access the same fragments. From level to level, the number of relevant fragments and queries decreases. The individual final chunks of step 1 (i.e., chunk 1 and chunk 2 in Figure 5.4) are the input of the next step.

  In Figure 5.4, we use a single iteration to split the workload into two chunks. The top node represents the total (undivided) workload, characterized by $N = 10$ relevant fragments and $Q = 5$ relevant queries. After the split, in chunk 1 and chunk 2, there are only three relevant queries per chunk.

  Compared to the optimal model that considers node failures, in step 1, we reduce the problem complexity by using a decomposition and not taking node failures into account.

- **Step 2: Adding Robustness on the Final Decomposition Level.** Based on the chunks on the final level of step 1, step 2 derives robust allocations for

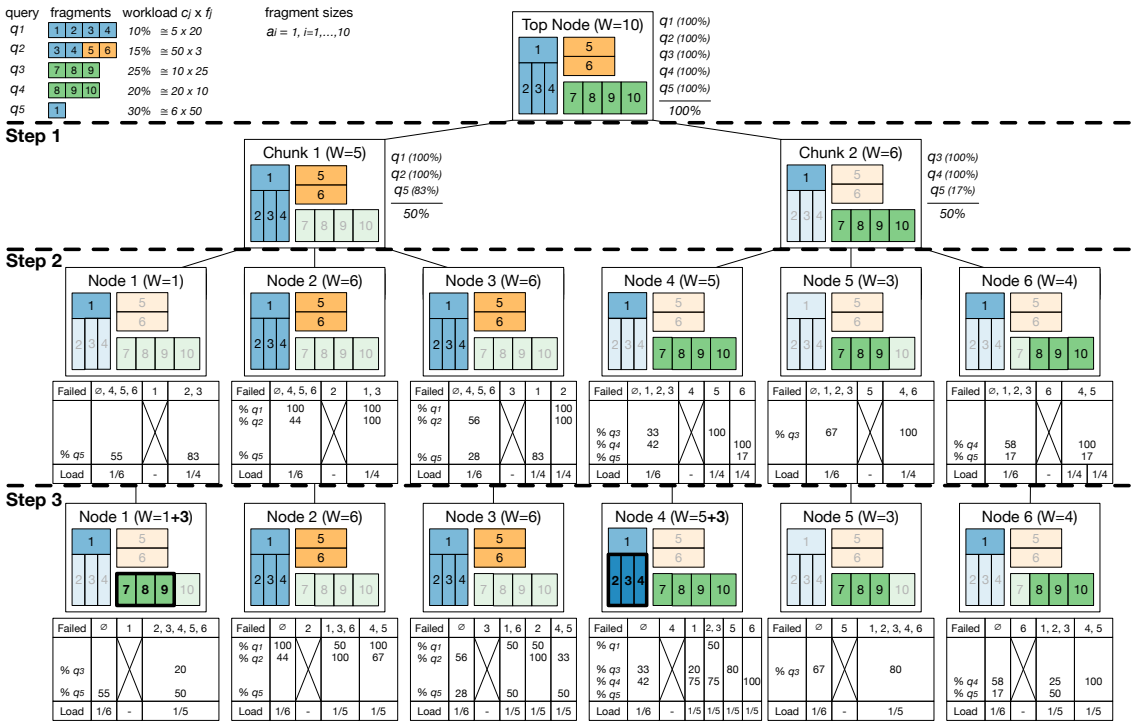


Figure 5.4.: Decomposition-based three-step heuristic approach that considers **node failures**: splitting the workload into chunks in step 1; compensating potential node failures within chunks in step 2; optimal fragment enhancement in step 3; illustration for $K = 6$ nodes, $B = 2$ chunks, a regular workload limit $L = 1/6$ and a guaranteed worst-case workload limit of $L_{\max}^{(-)} = 1/5$.

nodes of individual chunks *independently*: We use the optimal model that considers node failures and solve one ILP for each chunk with smaller inputs (i.e., fewer fragments, queries, and nodes) compared to the overall workload.

Figure 5.4 shows an example of step 2 with two chunks. Besides a regular workload distribution (cf. ∅), the solution of step 2 provides K individual emergency load distributions for each potential node failure $k^{(-)} = 1, ..., K$. After step 2, the allocation is such that regarding a single node (for example, node 1), the emergency load distributions are only affected for node failures of the same chunk ($k^{(-)} = 2, 3$). 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 2 (load 1/4). The other chunk's nodes (i.e., node 4 - 6) retain their regular workload distribution (load 1/6). Note, the allocation after step 2 *does not guarantee an optimal/perfect load balancing in failure cases.*

The chunk-based approach has properties that naturally support robust optimization approaches. Since the chunking clusters queries that use similar fragments, the need for additional data to guarantee the workability in case of

failures within one chunk is comparatively small. For example, in Figure 5.4, the replication factor after step 2 (with increased robustness) is $W/V = 2.5$; Figure 5.5 (see step 1) shows the (non-robust) basic solution, which has the replication factor $W/V = 2.1$.

Compared to the optimal model that considers node failures, in step 2, we reduce the complexity by applying the model with a smaller number of fragments, queries, and nodes.

- **Step 3: Fragment Enhancements for an even Load Balancing in Failure Cases.** In the final step 3, we 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 $x_{i,k}^{(f)} \in \{0,1\}$ denote the fragment allocation derived in step 2, i.e., whether fragment i is assigned to node k ($x_{i,k}^{(f)} = 1$) or not ($x_{i,k}^{(f)} = 0$), $i = 1,...,N, k = 1,...,K$. For example, for node $k = 6$, $x_{i,6}^{(f)} = 1$ for $i = 1, 8, 9, 10$ in Figure 5.4. We consider this allocation as *fixed*, i.e., these fragments are also assigned to the nodes in the final allocation. In the following ILP, we use binary decision variables $x_{i,k}^{(e)} \in \{0,1\}$ to decide whether to add fragment i to node k:

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

$$1/V \cdot \sum_{i=1,...,N,k=1,...,K} a_i \cdot x_{i,k}^{(e)} + \alpha \cdot L^{(-)} + \alpha/100 \cdot L \tag{5.20}$$

subject to (5.13) - (5.19) (i.e., the constraints of the optimal model considering node failures) for given $x_{i,k}^{(f)} \in \{0,1\}$ (i.e., the solution after step 2) where

$$x_{i,k}^{(f)} + x_{i,k}^{(e)} = x_{i,k}, \quad i = 1,...,N, k = 1,...,K \tag{5.21}$$

and an *optional* constraint, in case the additional enhancements shall not exceed a given memory budget $E \cdot V$, $E \geq 0$,

$$1/V \cdot \sum_{i=1,...,N,k=1,...,K} a_i \cdot x_{i,k}^{(e)} \leq E. \tag{5.22}$$

The ILP model of step 3 is similar to the optimal model that considers node failures. However, it is of much lower complexity because $x_{i,k}^{(f)}$ are given parameters (see Section 5.2.2), i.e., these fragments cannot be removed. Further, the freedom of the enhancement variables $x_{i,k}^{(e)}$ is limited because the

constraints (5.21) imply $x_{i,k}^{(e)} = 0$ for all $x_{i,k}^{(f)} = 1$, $i = 1, ..., N, k = 1, ..., K$. Moreover, the allocation $x_{i,k}^{(f)}$ provides a memory-efficient starting solution with a reliably good emergency workload limit $L^{(-)}$, which we discuss next (see performance guarantees).

In step 3, comparably little data has to be assigned in total because the allocations $x_{i,k}^{(f)}$ 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, for multiple chunks, it is sufficient when each chunk can take and pass enough load to *one* other chunk, and all chunks are connected (see Section 4.3.1, chaining approach for node failures).

In Figure 5.4, the enhancements of step 3 are visualized by framed/highlighted fragments, which are added to the allocation of step 2. While to node 1, fragment 7 - 9 are added, node 4 is completed by fragment 2 - 4. Due to the enhancements, after step 3, we obtain that, whatever node fails, a perfect workload distribution can always be achieved ($L^{(-)} = 1/5$). The final replication factor $W^F/V = 3.1$ is close to the optimal solution $W^{F*}/V = 2.8$ (see Figure 5.3).

Compared to the optimal model that considers node failures, in step 3, we reduce the complexity by building on a comparably robust basis solution with fixed fragments.

We summarize the interplay of the three steps: Step 1 reduces the complexity of the initial problem using a memory-efficient workload decomposition. Exploiting the optimal model for node failures, 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 over all nodes using optimal data enhancements.

Within step 2, computation time and worst-case limits can be balanced using smaller or larger chunk sizes. Naturally, for large chunks, computation times can become large because we apply the optimal model for node failures.

We can also control the complexity of the final data enhancement: (i) We can apply step 3 to *subsets/groups of chunks* first (e.g., when K is large). (ii) We can decrease the workload limit $L^{(-)}$ via an additional constraint in multiple steps until the optimal limit is reached. Suboptimal workload limits reduce the minimally required data enhancements and thus the solver's viable solution space for reaching the load balancing goal.

Performance Guarantees. Allocations after step 3 guarantee an even load balancing. In the following, we discuss the load balancing guarantees for allocations

after step 2. As step 2 is based on the decomposition-based heuristic, which ensures an even load balancing without node failures, its results concerning the worst-case limits can be calculated analytically. The remaining nodes of an affected chunk on the *final* level with n_b leaves end up with a worst-case limit of $1/(n_b - 1) \cdot n_b/K$. All other chunks are not affected by the node failure and, in turn, retain their workload limit of $1/K$. It follows that the highest workload share $L_{max}^{(-)}$ that can occur after step 2 is

$$L_{max}^{(-)} := \max_{b=1,...,B} \{n_b/(n_b - 1)\} /K. \tag{5.23}$$

Hence, the $L_{max}^{(-)}$ value is better (smaller) the larger the number of nodes n_b are chosen on the final level and depends on the minimum $n_{\min} := \min_{b=1,...,B}\{n_b\}$. The difference $L_{\max}^{(-)} - L^{(-)*} = n_{\min}/(n_{\min} - 1)/K - 1/(K - 1)$ to the optimal solution decreases in K and $n_{\min}$.

The solutions after step 2 allow giving performance guarantees. For a given number K, the chunking can be chosen such that a targeted worst-case workload share $L_{max}^{(-)}$ (see (5.23)) in the case of an arbitrary node failure is guaranteed. To effectively balance computation time and performance results, it is advisable to use a small number of chunks next to the tree's root and large chunks for the tree's leaves. Finally, based on the results of step 2, the ILP model of step 3 guarantees optimal (memory-efficient) fragment enhancements with a perfect load balancing of $L^{(-)} = 1/(K - 1)$.

Two-Step Approach. For the three-step approach, base allocations $x_{i,k}^{(f)}$ are a prerequisite for the applicability of step 3 enhancements because without a suitable backbone solution, the ILP model for the optimal data enhancements can be too complex: To an extreme, without fixing any fragments (i.e., $x_{i,k}^{(f)} = 0$ for all $i = 1, ..., N, k = 1, ..., K$), step 3 corresponds to the optimal model that considers node failures. Using more or less flexible (but optimized) base allocations, we can trade calculation time for memory consumption: Fixing fewer fragments to nodes (e.g., starting with a less robust base allocation) gives the solver more optimization potential. Naturally, the increasing flexibility increases the computation time.

One specific approach, called two-step approach, uses a basic decomposition-based allocation as input for the optimal fragment enhancement. The approach is visualized in Figure 5.5. Compared to the three-step approach in Figure 5.4, the two-step's overall memory consumption is lower ($W/V = 2.9$ vs. $W/V = 3.1$), but the final data enhancement is more complex, which is indicated by the lower ($21 < 25$) number of fixed fragments before the data enhancement and the higher ($8 > 6$) number of added fragments during the data enhancement.

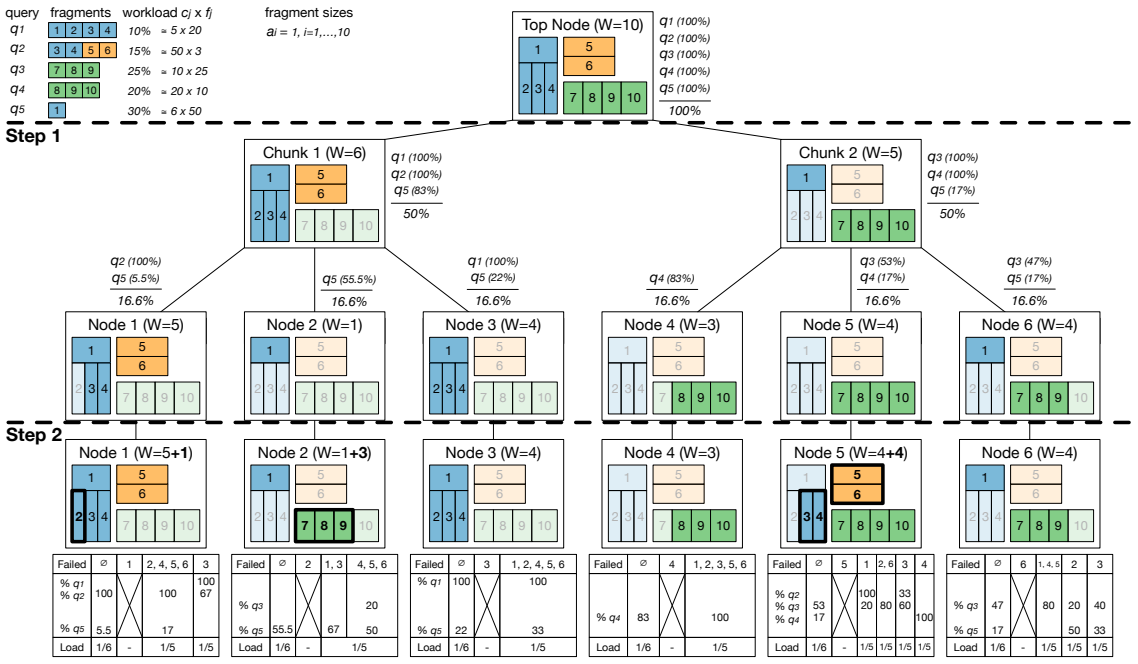


Figure 5.5.: Decomposition-based two-step heuristic approach that considers **node failures**: splitting the workload into chunks in step 1; compensating potential node failures with optimal fragment enhancements in step 2; illustration for $K = 3 + 3 = 6$ nodes, a regular workload limit $L = 1/6$ and a guaranteed worst-case workload limit of $L_{\max}^{(-)} = 1/5$.

5.3.2. Workload Uncertainty

Workloads are usually not perfectly predictable: (i) For example, query frequencies may fluctuate over time. (ii) Query costs (e.g., query execution times) may deviate from the modeled values. (iii) Finally, unexpected query classes may be sent to the database system.

Instead of calculating an allocation that is optimized for a *single workload scenario*, characterized by a set of query classes j with their frequencies f_j and costs c_j, $j = 1, ..., Q$, we can also use multiple scenarios as model input. The core idea is finding a single, more robust allocation that enables an even load balancing for these multiple potential workload scenarios (by assigning additional data fragments to nodes). By using multiple scenarios as model input, we can ensure load balancing robustness against multiple *known* but also *unknown* workload scenarios:

- **Known Workload Scenarios.** If potential future workloads are known, we can use them directly as allocation input. With an increasing number of input scenarios, the memory consumption increases.

- **Unknown Workload Scenarios.** Not all potential future workload scenarios may be known, or there may be too many of them. However, by using spe-

Table 5.4.: Notation table for ILP models that consider workload uncertainty.

symbol	description
N	number of fragments $i, i = 1, ..., N$
Q	number of queries $j, j = 1, ..., Q$
K	number of nodes $k, k = 1, ..., K$
$\boldsymbol{S}$	number of workload scenarios $s, s = 1, ..., S$
a_i	size of fragment $i, i = 1, ..., N$
q_j	accessed fragments of query j, $j = 1, ..., Q$, a subset of {1,...,N}
$\boldsymbol{f_{j,s}}$	frequency of query j for scenario s, $j = 1, ...Q, s = 1, ..., S$
c_j	costs of processing query $j, j = 1, ..., Q$
$\boldsymbol{C_s}$	total workload costs for scenario s, s=1,...,S
V	overall size of all accessed fragments
α	penalty factor for the workload limit
$x_{i,k}$	fragment i is allocated to node k: yes (1) / no (0)
$y_{j,k}$	query j can run on node k: yes (1) / no (0)
$\boldsymbol{z_{j,k,s}}$	query j's workload share assigned to node k for scenario s: $\in [0, 1]$
$\boldsymbol{W^U}$	memory consumption of an ILP approach considering workload uncertainty
L	a node's workload limit

cific diversified input scenarios, the resulting allocation also allows improved load balancing for *unseen* scenarios that are similar to mixtures of input scenarios. In general, an allocation's robustness can be increased by choosing a larger number of diverse input scenarios. Thereby, we can smoothly choose between allocations that are optimized against a few scenarios with a small memory consumption and full replication, which is naturally robust against all random workloads.

Optimal Model

In the following, we consider S potential workload scenarios. We use Q as the overall number of potential query classes over all scenarios. Each scenario s, $s = 1, ..., S$, is characterized by query frequencies $f_{j,s}$ and associated workload costs $C_s := \sum_{j=1,...,Q} f_{j,s} \cdot c_j$. 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. We want to find a single allocation that enables an even load balancing for all S potential workload scenarios. Compared to the basic model, we use extended variables for the workload share $z_{j,k,s}$ per scenario s, $s = 1, ..., S$. Table 5.4 summarizes the input parameters, decision variables, and solution variables for our extension that considers workload uncertainty. New model parameters and variables are highlighted.

In the following, we present our ILP formulation to calculate allocations that are optimized for multiple potential workloads:

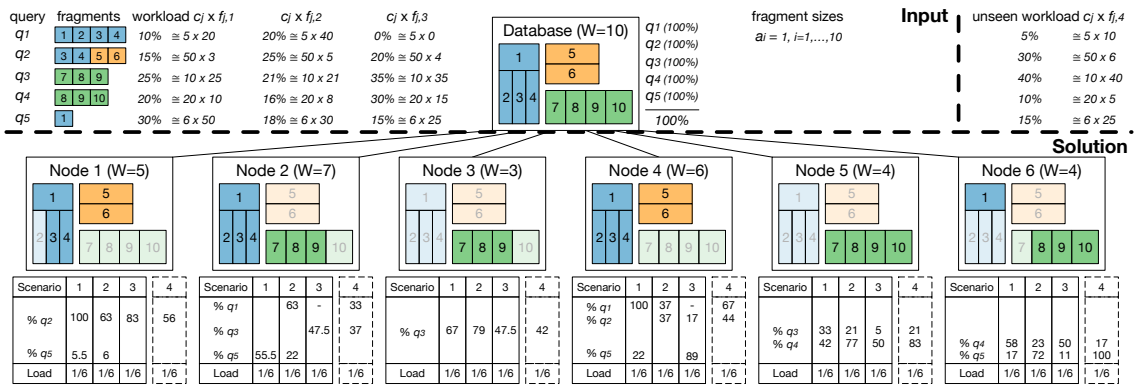


Figure 5.6.: Optimal solution that considers **multiple workload scenarios**: illustration for $S = 3$ input scenarios and $K = 6$ nodes. An optimal workload limit $L^* = 1/6$ for all input scenarios is ensured. Thereby, the allocation can also evenly distribute the unseen scenario.

$$
\begin{aligned}
&minimize \\
&x_{i,k}, y_{j,k} \in \{0,1\}, z_{j,k,s} \in [0,1], 0 \leq L \leq 1, \\
&i = 1,...,N, j = 1,...,Q, k = 1,...,K, s = 1,...,S
\end{aligned}
$$

$$
1/V \cdot \sum\nolimits_{i=1,...,N,k=1,...,K} x_{i,k} \cdot a_i + \alpha \cdot L \tag{5.24}
$$

subject to constraints for multiple workload scenarios:

$$
y_{j,b} \cdot |q_j| \leq \sum\nolimits_{i \in q_j} x_{i,k}, \qquad j = 1,...,Q, k = 1,...,K \tag{5.25}
$$

$$
z_{j,k,s} \leq y_{j,k}, \qquad j = 1,...,Q, k = 1,...,K, s = 1,...,S \tag{5.26}
$$

$$
\sum\nolimits_{j=1,...,Q} f_{j,s} \cdot c_j / C_s \cdot z_{j,k,s} \leq L, \qquad k = 1,...,K, s = 1,...,S \tag{5.27}
$$

$$
\sum\nolimits_{k=1,...,K} z_{j,k,s} = 1, \qquad j = 1,...,Q, s = 1,...,S \tag{5.28}
$$

The objective (5.24) minimizes the replication factor W/V and contains a penalty term for the largest workload share L. Note, in contrast to the extension that considers node failures, we only need a single load limit (as in the basic model) because the number of nodes per scenario is fixed (i.e., K).

The constraints (5.25) correspond to the constraints (5.2) of the basic model, i.e., they guarantee that a query j can only be executed at node k if all relevant fragments are available. The constraints (5.26) - (5.28) correspond to the constraints (5.3) - (5.5) of the basic model but consider multiple workload scenarios. Overall, the ILP model (5.24) - (5.28) *simultaneously* enables an even load balancing for S workload scenarios. Hence, it is a generalization of the basic model, which only covers the special case $S = 1$.

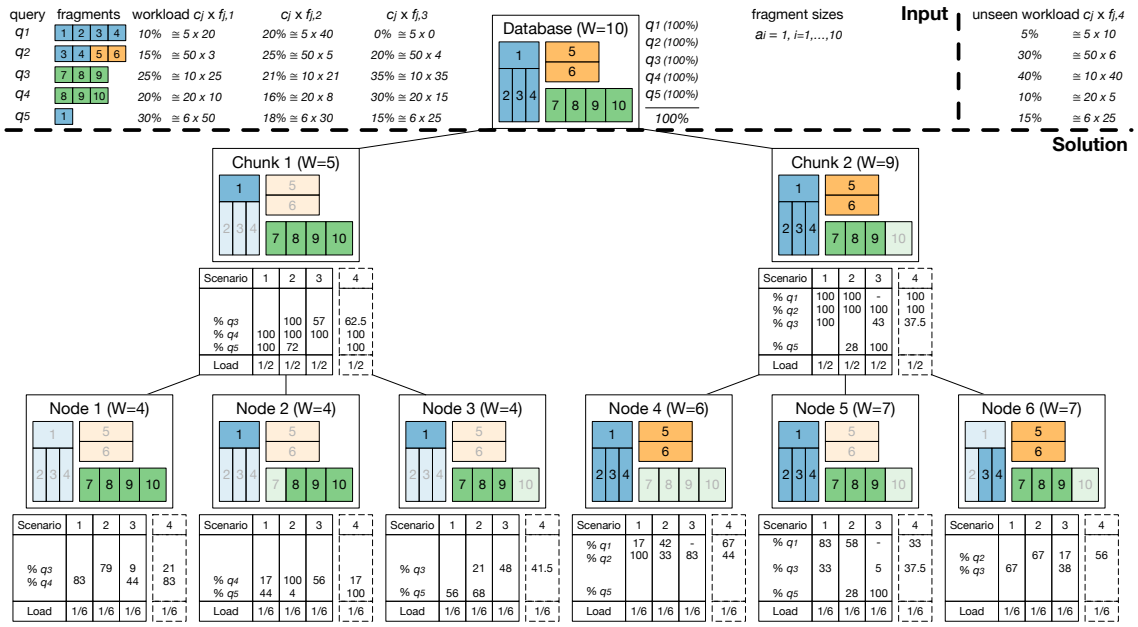


Figure 5.7.: Decomposition-based heuristic approach that considers **multiple workload scenarios**: illustration for $S = 3$ input scenarios and $K = 6$ nodes. An optimal workload limit $L^* = 1/6$ for all input scenarios is ensured. Thereby, the allocation can also evenly distribute the unseen scenario.

Figure 5.6 shows an allocation for $S = 3$ input scenarios. While minimizing the overall memory consumption, the solution enables an even load balancing (i.e., $L = 1/6$) for the three input scenarios. In the tables of Figure 5.6, the values $z_{j,k,s}$ are shown in the columns 1 - 3. For example, the workload share of query $j = 2$ in scenario $s = 3$ is split among the nodes $k = 1$ ($z_{2,1,3} = 83\%$) and $k = 4$ ($z_{2,4,3} = 17\%$). We can also evaluate the load balancing for unseen workload scenarios, which are not used as model input, e.g., the load of scenario $\tilde{s} = 4$ could be balanced evenly (see the columns 4 in Figure 5.6).

Decomposition-Based Approach

The presented model extension for multiple workload scenarios is compatible with our decomposition-based heuristic (see Section 5.1.2): We can split the workload iteratively using our extended model (5.24) - (5.28). Again, the relevant fragments and queries decrease from level to level. In contrast to the basic decomposition approach, each chunk enables S workload distributions. The individual workload splits are, thus, of a higher complexity for the same number of subnodes. Naturally, we can choose a smaller number of subnodes B until it is minimal (i.e., $B = 2$). We can also use the presented approaches to lower the computation time (see Section 5.2). In particular, for our clustering approaches if $S > 1$, we order the queries according to their *expected workload* over all scenarios $s = 1, ..., S$, i.e., $c_j/S \cdot \sum_{s=1,...,S} f_{j,s}$.

Table 5.5.: Notation table for ILP extensions that consider data modification costs.

symbol	description
N	number of fragments $i, i = 1, ..., N$
Q	number of (read) queries $j, j = 1, ..., Q$
$\boldsymbol{U}$	number of update queries $\bar{j}, \bar{j} = 1, ..., U$
K	number of nodes $k, k = 1, ..., K$
f_j	frequency of (read) query $j, j = 1, ...Q$
c_j	costs of processing (read) query $j, j = 1, ..., Q$
$\bar{\boldsymbol{m}}_{\bar{\boldsymbol{j}}}$	fragments that are modified by update query $\bar{j}, m_{\bar{j}} \subseteq \{1, ..., N\}$
$\bar{\boldsymbol{q}}_{\bar{\boldsymbol{j}}}$	fragments that are only accessed by update query $\bar{j}, \bar{q}_{\bar{j}} \subseteq \{1, ..., N\} \setminus m_{\bar{j}}$
$\bar{\boldsymbol{f}}_{\bar{\boldsymbol{j}}}$	frequency of update query $\bar{j}$
$\tilde{\boldsymbol{c}}_{\bar{\boldsymbol{j}},\boldsymbol{i}}$	fragment modification costs of update query $\bar{j}$ for fragment i
$\bar{\boldsymbol{c}}_{\bar{\boldsymbol{j}}}$	data retrieval costs of update query $\bar{j}$
C	total (read) workload costs
α	penalty factor for the workload limit
$x_{i,k}$	fragment i is allocated to node k: yes (1) / no (0)
$y_{j,k}$	query j can run on node k: yes (1) / no (0)
$z_{j,k}$	query j's workload share assigned to node k: $\in [0, 1]$
$\boldsymbol{Y}_{\bar{\boldsymbol{j}},\boldsymbol{k}}$	update query $\bar{j}$ must be executed on node k: yes (1) / no (0)
$\boldsymbol{W}^{\boldsymbol{M}}$	memory consumption of an ILP approach considering modification costs
L	a node's workload limit

5.3.3. Data Modifications

Data fragments can change over time as a result of insert, delete, and update statements. Then, we have to update every node that holds these fragments. Data synchronization comes along with costs, which we must take into account for an even load balancing. In the following, we discuss how we can consider data modification costs in our approach.

Optimal Model

Similar to the read queries $j = 1, ..., Q$, we consider a set of U data modifying (hereinafter called simply "update") queries $\bar{j}$, characterized by modified $\bar{m}_{\bar{j}}$ and potentially only accessed $\bar{q}_{\bar{j}}$ fragments, i.e., $\bar{m}_{\bar{j}} \subseteq \{1, ..., N\}$, $\bar{q}_{\bar{j}} \subseteq \{1, ..., N\} \setminus \bar{m}_{\bar{j}}$, $\bar{j} = 1, ..., U$. Update queries $\bar{j}$ occur with frequency $\bar{f}_{\bar{j}}$ and account for fragment modification costs $\tilde{c}_{\bar{j},i}$ and optional data retrieval costs $\bar{c}_{\bar{j}}$. Table 5.5 summarizes the input parameters, decision variables, and solution variables that we use to describe our extensions that consider data modification costs. New model parameters and variables are highlighted.

We can synchronize data replicas logically or physically (see Section 2.2.3). Following, we describe how we can extend our models to consider logical and physical synchronization costs.

- **Logical synchronization.** In this case, a primary node executes all update queries. Replica nodes are synchronized by re-executing relevant update queries, which modify stored fragments. To control which update query has to be executed at which node, we use additional binary decision variables $Y_{\bar{j},k}$, controlling if an update query $\bar{j}$ must be executed on node k ($Y_{\bar{j},k} = 1$) or not ($Y_{\bar{j},k} = 0$). Further, we extend our optimal model (5.1) - (5.5) with the following constraints:

$$\sum_{i \in \bar{m}_{\bar{j}}} x_{i,k} \leq Y_{\bar{j},k} \cdot |\bar{m}_{\bar{j}}| \quad \bar{j} = 1, ..., U, k = 1, ..., K \tag{5.29}$$

  The constraints (5.29) ensure that if a modified fragment $i \in \bar{m}_{\bar{j}}$ is stored (i.e., $x_{i,k} = 1$), the corresponding update query $\bar{j}$ must be executed (i.e., $Y_{\bar{j},k} = 1$). Thus, all modified and accessed fragments of the corresponding update query (i.e., $i \in \bar{m}_{\bar{j}} \cup \bar{q}_{\bar{j}}$) must be stored, which is ensured by the following constraints:

$$\sum_{i \in \bar{m}_{\bar{j}} \cup \bar{q}_{\bar{j}}} x_{i,k} \geq Y_{\bar{j},k} \cdot (|\bar{m}_{\bar{j}}| + |\bar{q}_{\bar{j}}|) \quad k = 1, ..., K, \bar{j} = 1, ..., U \tag{5.30}$$

  Finally, we have to extend the constraints (5.4) of the optimal model to include the costs for update queries: We add a cost term that summarizes the query costs for all required update queries $\bar{j}$, i.e.,

$$\sum_{j=1,...,Q} \frac{f_j \cdot c_j}{C} \cdot z_{j,k} + \sum_{\bar{j}=1,...,U} Y_{\bar{j},k} \cdot \frac{\bar{f}_{\bar{j}}}{C} \cdot (\bar{c}_{\bar{j}} + \sum_{i \in \bar{m}_{\bar{j}}} \tilde{c}_{\bar{j},i}) \leq L \quad k = 1, ..., K. \tag{5.31}$$

  If desired, we could further enforce that a single primary node (e.g., $k = 1$) executes all update queries (see constraint (5.32)).

- **Physical synchronization.** In this case, a primary node executes all update queries and propagates relevant fragment changes to all other nodes. These other nodes, then, update their stored fragments accordingly.

  For extending our optimal model (5.1) - (5.5) to consider physical data modification costs, we replace the (family of) constraints (5.4) with a special constraint for the primary node (see constraint (5.32)) and constraints for the replica nodes (see constraints (5.34)). In the following, we assume a single dedicated primary node $k = 1$. At the primary node, fragment modification costs and data retrieval costs occur for all update queries:

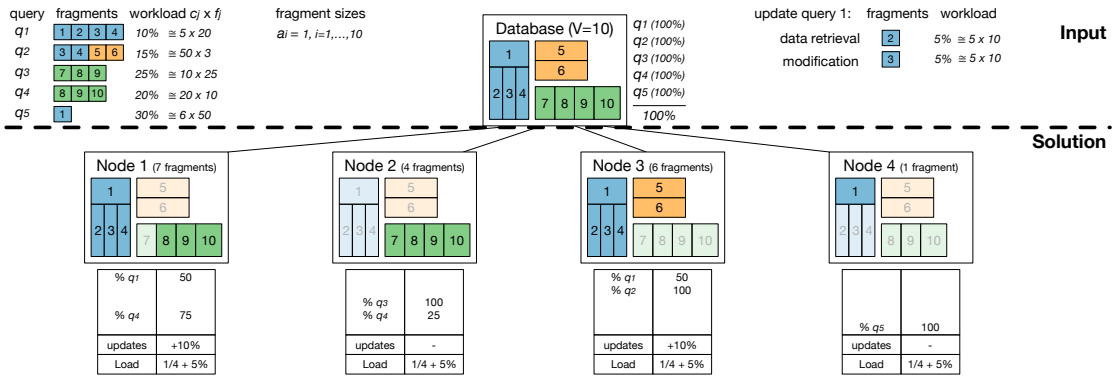


Figure 5.8.: Optimal solution that considers **logical modification costs**.

$$\sum_{j=1,...,Q} \frac{f_j \cdot c_j}{C} \cdot z_{j,1} + \sum_{\bar{j}=1,...,U} \frac{\bar{f}_{\bar{j}}}{C} \cdot (\bar{c}_{\bar{j}} + \sum_{i \in \bar{m}_{\bar{j}}} \tilde{c}_{\bar{j},i}) \leq L. \tag{5.32}$$

Further, the primary node must store all fragments for update queries:

$$\sum_{i \in \bar{m}_{\bar{j}} \cup \bar{q}_{\bar{j}}} x_{i,1} = |\bar{m}_{\bar{j}} \cup \bar{q}_{\bar{j}}| \quad \bar{j} = 1, ..., U. \tag{5.33}$$

For the other nodes $k = 2, ..., K$, only fragment modification costs $\tilde{c}_{\bar{j},i}$ for stored fragments ($x_{i,k} = 1$) have to be considered:

$$\sum_{j=1,...,Q} \frac{f_j \cdot c_j}{C} \cdot z_{j,k} + \sum_{\bar{j}=1,...,U} \frac{\bar{f}_{\bar{j}}}{C} \cdot \sum_{i \in \bar{m}_{\bar{j}}} \tilde{c}_{\bar{j},i} \cdot x_{i,k} \leq L \quad k = 2, ..., K. \tag{5.34}$$

For both synchronization approaches, the total workload including updates may exceed the overall read-only query costs C. Hence, the optimal workload limit L may also exceed $1/K$. It is difficult to determine the optimal workload limit because it is unknown how often each update query has to be executed at final nodes or how often each fragment is replicated. In this case, using the penalty approach with a penalty term $\alpha \cdot L$ for the maximum workload share L in the objective is preferable. This way, we reduce L as far as possible (with a large α) for low synchronization costs with an even load balancing or trade a lower memory consumption for higher synchronization costs (with a smaller α).

Figure 5.8 shows an allocation that considers costs for logical synchronization. The update query $\bar{j} = 1$ is executed at node 1 and node 3. The additional costs for update queries ($2 \cdot 10\%$) are balanced by higher read-only loads for the other nodes. Overall, the load per node is increased by ($20\%/4 =$)5%.

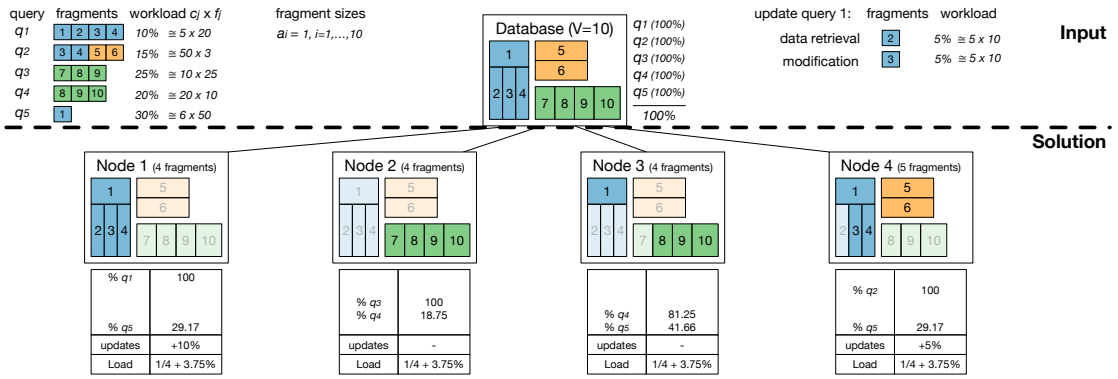


Figure 5.9.: Optimal solution that considers **physical modification costs**.

Figure 5.9 shows an allocation that considers costs for physical synchronization. The update query $\bar{j} = 1$ is executed at the primary node $k = 1$ for +10% execution costs. In addition, the modified fragment $i = 3$ is stored at node $k = 4$, which requires +5% fragment modification costs. As for physical synchronization, the additional costs for update queries (10% + 5%) are balanced with the read-only load. Overall, the load per node is increased by (15%/4 =)3.75%.

Decomposition-Based Approach

In our decomposition heuristic, the consideration of update queries is also possible. In the final level (considering only leaf nodes), query update costs can be accurately included as in the optimal model. However, in the intermediate levels, eventually occurring query update costs (i.e., for logical synchronization: at how many final nodes an update query has to be executed, and for physical synchronization: how often a modified fragment is replicated) are not known. As in many real-world applications the ratio between update costs and total workload costs is small [125], this inaccuracy is tolerable. However, to further minimize uneven workload distributions caused by ignoring update queries for write-intensive workloads, we can modify the constraints (5.31), (5.32), and (5.34) as follows: To account for the described uncertainty, a parameter $u_{\bar{j},b} \leq n_b$ (or $u_{i,b} \leq n_b$) can be used to approximate how many leaf nodes of chunk b are affected by the update costs for query $\bar{j}$ (or fragment i). To estimate expected numbers $u_{\bar{j},b}$ (or $u_{i,b}$), we can take fragment frequencies and the number of targeted nodes n_b into account.

5.3.4. Reallocation Costs

Workloads typically change over time. As a result, current optimized fragment allocations and workload distributions must be updated by (i) reallocating data fragments or (ii) adding or removing nodes. Because the reorganization of data is usually time-consuming and, thus, costly, we want to consider the reallocation effort.

Table 5.6.: Notation table for ILP extensions that consider data reallocation costs.

symbol	description
N	number of fragments $i, i = 1, ..., N$
Q	number of queries $j, j = 1, ..., Q$
K	number of nodes $k, k = 1, ..., K$
$\boldsymbol{A}$	number of added nodes $1 \leq A < K$
$\boldsymbol{R}$	number of removed nodes $1 \leq R < K$
a_i	size of fragment $i, i = 1, ..., N$
f_j	frequency of query $j, j = 1, ...Q$
c_j	costs of processing query $j, j = 1, ..., Q$
C	total (read) workload costs
α	penalty factor for the workload limit
$\boldsymbol{s_{i,k}}$	allocation state: fragment i is currently allocated at node k
$\boldsymbol{r_i}$	costs when adding fragment i to node k
$x_{i,k}$	fragment i is allocated to node k: yes (1) / no (0)
$y_{j,k}$	query j can run on node k: yes (1) / no (0)
$z_{j,k}$	query j's workload share assigned to node k: $\in [0, 1]$
X_k	node k is used for query execution: yes (1) / no (0)
$\boldsymbol{W^R}$	memory consumption of an ILP approach considering reallocation costs

Reallocation goals and costs may depend on specific systems, the data partitioning, and other factors. For example, one may want to minimize the added fragments per node *or* the overall added fragments. Further, the reallocation of single fragments may require the reorganization of complete tables (e.g., adding columns to row stores). A survey of possible reallocation costs and reallocation goals is not part of this thesis. In the following, we show one exemplary model that considers reallocation costs based on an existing allocation.

Optimal Model

We assume a current *allocation state* characterized by the parameters $s_{i,k} \in \{0, 1\}$, which indicate whether fragment i is placed at node k ($s_{i,k} = 1$) or not ($s_{i,k} = 0$), $i = 1, ..., N$, $k = 1, ..., K$. Further, we assume reallocation costs r_i (if a new fragment i is transferred to node k, $i = 1, ..., N$) that are *independent* of the allocation state of the other fragments. In case the data of fragment i is deleted at node k, we assume *no/negligible costs*. Table 5.6 summarizes the input parameters, decision variables, and solution variables that we use to describe our extensions that consider data reallocation costs. New model parameters and variables are highlighted.

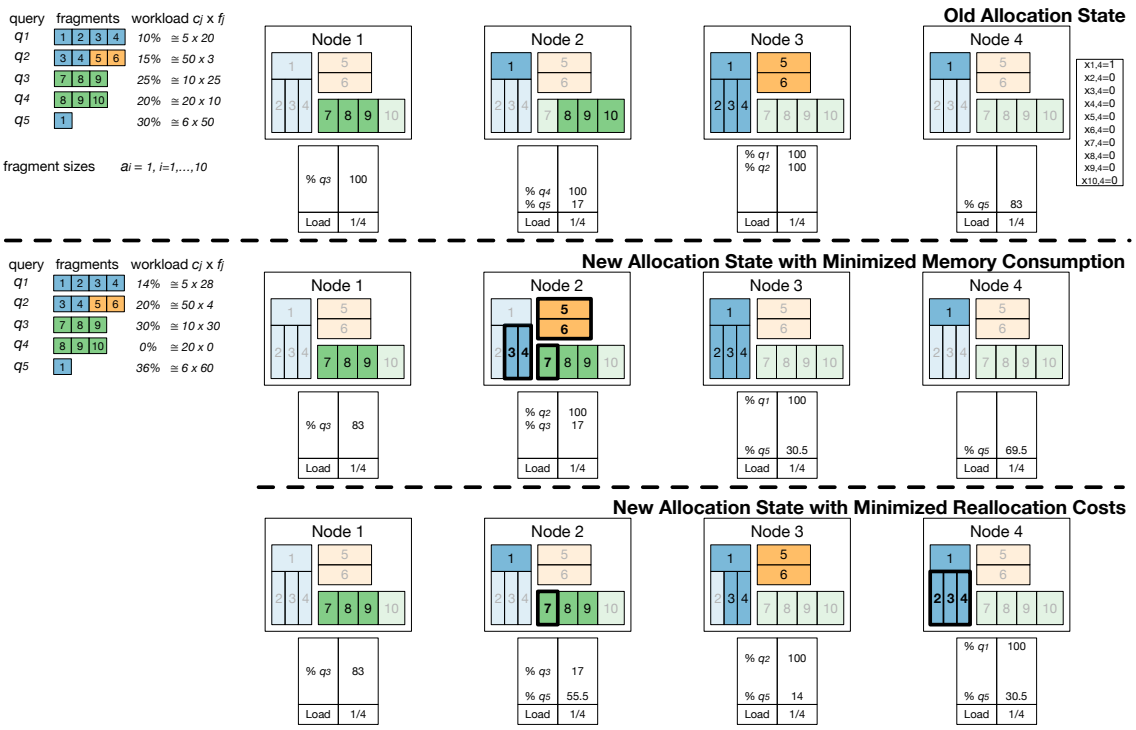


Figure 5.10.: Optimal solutions that consider **data reallocation costs**.

To consider reallocation costs and reduce the amount of overall added data, the objective of the optimal model (5.1) - (5.5) is extended by the cost term

$$\sum_{i=1,\ldots,N,k=1,\ldots,K} r_i \cdot (1 - s_{i,k}) \cdot x_{i,k}. \tag{5.35}$$

If fragment i is already allocated to node k (i.e., $s_{i,k} = 1$), there are no reallocation costs independently of $x_{i,k}$ because $(1 - s_{i,k}) \cdot x_{i,k} = 0$. If fragment i is *not* allocated to node k (i.e., $s_{i,k} = 0$), there are reallocation costs only if fragment i gets allocated to node k (i.e., $x_{i,k} = 1$) because $(1 - s_{i,k}) \cdot x_{i,k}$ depends on $x_{i,k}$.

The cost term prevents that large parts of the data are reorganized while the benefit concerning the overall memory consumption (compared to solutions that do not consider reallocation costs) is only marginal. In the new objective term

$$\sum_{i=1,\ldots,N,k=1,\ldots,K} x_{i,k} \cdot (a_i + r_i \cdot (1 - s_{i,k}))$$

we can prioritize a low memory consumption ($a_i > r_i$) or low reallocation costs ($a_i < r_i$). We could also trade reallocation costs vs. load balancing: We can accept a slight load imbalance to lower reallocation costs (or the memory consumption).

In Figure 6.13, we show two new allocations, which are based on an old allocation state and consider reallocation costs. Fragments that are added during the

reallocation are highlighted with a bold frame. When prioritizing a low memory consumption (see the middle allocation), we must add 5 fragments (all to node 2) and achieve an allocation with overall $15 = 3 + 7 + 4 + 1$ fragments. When prioritizing low reallocation costs (see the bottom allocation), we must add fewer (4) fragments (to node 2 and 4) but achieve an overall higher ($16 = 3 + 4 + 5 + 4$) memory consumption.

Adding or Removing Nodes. Based on this model extension, we can also calculate optimized allocations with a changed number of nodes, e.g., for elastic systems [151, 207]. When adding a number of nodes $A, 1 \leq A < K$, without stored fragments, we set $s_{i,k} = 0, i = 1, ..., N, k = K - A + 1, ..., K$. When removing a number of nodes $R, 1 \leq R < K$, with stored fragments, we could simply calculate new allocations without *all subsets of nodes* (because we do not know which node removal offers the greatest optimization potential) independently and take the one with the lowest reallocation costs. Because the number of node combinations increases quickly with the number of nodes to remove R, we can also rely on the solver's efficiency and integrate the decision of which node(s) to remove into our ILP model. Therefore, we introduce new decision variables X_k, which control whether a node remains in the cluster ($X_k = 1$) or is removed ($X_k = 0$), $k = 1, ..., K$. Further, we add the following constraints:

$$X_k \geq y_{j,k}, \qquad j = 1, ..., Q, k = 1, ..., K \tag{5.36}$$

$$\sum_{k=1,...,K} X_k = K - R \tag{5.37}$$

The family of constraints (5.36) ensures that no query is executable at node k (i.e., $y_{j,k} = 0, j = 1, ..., Q$) if node k is not used for query execution (i.e., $X_k = 0$). The constraint (5.37) ensures that the number of used nodes is $K - R$ (i.e., R nodes are removed).

The reallocation of running nodes may degrade the query performance. When adding nodes, it could be desirable to not change the fragment allocation of running nodes. Instead, we could only extend the allocation by a node so that the load can still be balanced. The added nodes could later be removed, and an even load balancing would still be possible. Note, the existence of such an allocation is guaranteed because we can always scale a read-only workload by adding a full replica. The following family of constraints ensures that existing nodes are not reallocated when adding nodes:

$$x_{i,k} = s_{i,k}, \qquad i = 1, ..., N, k = 1, ..., K - A \tag{5.38}$$

Similarly, we could also allow that fragments of existing nodes can be removed but not added:

$$x_{i,k} \leq s_{i,k}, \qquad i = 1, ..., N, k = 1, ..., K - A \tag{5.39}$$

Decomposition-Based Approach

We can also include reallocation costs in our decomposition-based heuristic. When using intermediate allocations with chunks, there is a fixed relation between these chunks and associated final nodes. Thus, the allocation states $s_{i,k}$ of final nodes k can be uniquely translated to intermediate allocation states $s_{i,b}$ with chunks b. Hence, reallocation costs can be included in objective (5.6) of the generalized decomposition-based approach by using these chunk states $s_{i,b}$.

An existing allocation state restricts the possible new allocations with low reallocation costs. Because the new allocation must be similar to the old one (to have low reallocation costs), the corresponding optimization problem is often easier to solve than the basic model. Hence, if low reallocation costs are targeted, the limited search space enables (i) using the optimal model even for larger problems, (ii) using fewer decomposition steps with larger chunks, or (iii) faster calculations.

5.4. Summary

Our workload distribution problem for partially replicated database clusters can be formulated as ILP program and, thus, solved optimally using state-of-the-art solvers. Because ILP is not scalable, larger problems eventually become intractable. Therefore, we propose three ILP-based heuristics, which can be flexibly combined to lower the calculation time: (i) a decomposition approach for iteratively splitting the workload, (ii) query clustering (in particular for a large number of queries with small workload shares), and (iii) solver relaxations via an optimality gap or time limit. Using ILP, we can also model varied model extensions, e.g., considering node failures, workload uncertainty, data modification costs, and reallocation costs. Besides presenting the optimal programming models, we discussed how to address the extensions using our heuristics, particularly the decomposition approach.

6

Evaluation of Allocation Models

Magic mirror in my hand, who is the fairest in the land?

Brothers Grimm

In this chapter, we evaluate our allocation models for various input parameters, covering the standard benchmarks TPC-H and TPC-DS as well as a real-world accounting workload. We compare our results against state-of-the-art techniques with regard to the memory consumption, workload distribution, and calculation time as well as the end-to-end throughput. We first describe our evaluation methodology in Section 6.1. Following, we evaluate our basic models for read-only workloads (Section 6.2) and model extensions (Section 6.3). We summarize our evaluation results in Section 6.4. Finally, we discuss limitations of our approach and evaluation in Section 6.5.

6.1. Methodology

An objective comparison of allocation approaches is challenging because the solution quality depends on the multifaceted input parameters: The complexity of an allocation cannot be easily derived by only the number of fragments, queries, and nodes. Of course, these numbers determine, for example, an upper limit of combinations for an exhaustive search. But also, the accessed fragments per query, query costs, query frequencies, and fragment sizes influence how well we can prune suboptimal solutions or how well we can heuristically allocate queries and fragments to nodes.

Overall, the best allocation approach may depend on the specific problem instance and, thus, the evaluation scenario. For this reason, we evaluate our allocation approaches and extensions for multiple input parameters. In the following, we describe our applied methodology, including (i) the selected database workloads and how we derived model inputs (Section 6.1.1), (ii) the calculation of allocations (Section 6.1.2), and (iii) the evaluation of allocations (Section 6.1.3).

6.1.1. Workloads

We evaluate our allocation models using three workloads of different complexities: the two standardized analytical benchmarks TPC-H and TPC-DS as well as one real-

world accounting workload. For all workloads, we use vertical partitioning with each column as individual fragment: For the accounting workload, we had no other choice because we got only access to metadata and could only derive accessed fragments on a columnar granularity. For TPC-H and TPC-DS (and any other SQL workload), using columnar partitioning allowed us to reliably and, thus, automatically derive the (potentially) accessed fragments/columns based on the SQL query.

The three workloads are characterized by smaller and larger numbers of fragments N and queries Q, resulting in different problem sizes. We focus our evaluations on cluster sizes with $2 \leq K \leq 16$ nodes, which correspond to the cluster sizes of our end-to-end evaluations. Optimal allocations naturally depend on fragment sizes a_i, query frequencies f_j, query costs c_j, and accessed fragments q_j. In the following, we explain how we derived all model inputs.

TPC-H and TPC-DS. We set up the benchmark tables with scale factor 10 in a cluster of commercial columnar in-memory database systems. The chosen database system and scale factor usually have only a small effect on the complexity of allocations because only the values for fragment sizes a_i and query costs c_j change. In previous evaluations [104, 195], we used PostgreSQL as database system and can, thus, compare our results for different parameters of the same workload.

The TPC-H benchmark comprises a schema with $N = 61$ columns and $Q = 22$ query templates. The more complex TPC-DS benchmark is characterized by $N = 425$ columns and $Q = 99$ query templates. We derived fragment/column sizes $a_i, i = 1, ..., N$, from the database catalog. For all queries, we set $f_j := 1, j = 1, ..., Q$, as default to model an identical number for each query template. We modeled query costs c_j as average processing time of query j with random template parameters, $j = 1, ..., Q$.

For TPC-DS, processing the queries 22, 36, 70, and 86 took exceptionally long, including timeouts for specific template parameters. Having a few queries dominating the overall processing time makes the allocation easier: In an extreme case, all but one replica process a single expensive query while a single replica processes the rest of the queries. Thus, we omitted these queries in our allocations by setting the query frequencies $f_j := 0, j = 22, 36, 70, 86$, resulting in $Q = 95$ included queries for TPC-DS.

Real-World Accounting Workload. We got access to metadata of an enterprise's central accounting table with $N = 344$ columns and a summary of a workload trace against this table in the form of $Q = 4461$ query templates and statistics. The metadata enabled us to derive all required model inputs (i.e., fragment sizes, accessed attributes per query, query frequencies, and average query processing times) for calculating fragment allocations using vertical (columnar) partitioning.

Table 6.1 summarizes the workloads used for the evaluation. All model inputs to reproduce the calculation of all allocations are available online [99].

Table 6.1.: Overview of workloads used for the evaluation.

Workload	Dataset	#Fragments N	#Queries Q (included)
TPC-H	Synthetic uniform	61	22
TPC-DS	Synthetic skewed	425	99 (95)
Accounting Workload	Real-world	344	4461

6.1.2. Calculation of Allocations

We calculated all allocations on a laptop with an Apple M1 Pro CPU with 8 cores and 32 GB RAM. For the calculation of all our ILP-based approaches, we implemented the models with AMPL [78] and solved them using Gurobi [94] (version 9.5.2). For the comparison with state-of-the-art approaches, we implemented the algorithms in Python 3, and declare the runtimes as an upper limit ($<$).

For obtaining the model inputs and running the end-to-end evaluations, we used a database cluster with 16 nodes. Each node is a virtual machine with four cores, 50 GB RAM, running SUSE Linux Enterprise Server 15 SP1 and a commercial columnar in-memory database system. The 16 virtual machines share four Xeon (Nehalem EX) X7560 CPUs with overall 32 physical and 64 ($= 16 \cdot 4$) virtual (using hyper-threading) cores, and 1024 GB RAM. Each virtual machine is pinned to 2 physical (4 virtual) cores. As benchmark client, we used a virtual machine with an Intel Core i7 9xx CPU with four cores and 4 GB RAM.

6.1.3. Evaluation of Allocations

The investigated allocation problem is a macro optimization, considering a workload in a potentially long period of time. We can evaluate allocations (i) numerically, i.e., with regard to the memory consumption W, (maximum) load limit/share L (over all nodes), and calculation time, as well as (ii) in end-to-end evaluations, i.e., how the calculated allocations perform when they are deployed on database servers running workloads in practice.

Numerical Evaluations

Numerical evaluations are limited by the calculation time. As heuristic solutions can often be calculated (relatively) quickly, we can conduct numerical evaluations using multiple models and settings (e.g., different chunkings for the decomposition approach, solver settings, and workload scenarios with regard to failed nodes and query frequencies). Only optimal solutions eventually become intractable because of too high calculation times.

Visualization of Allocations. We can also visualize allocation inputs and solutions to derive structural insights and to detect an algorithm's strengths as well as its po-

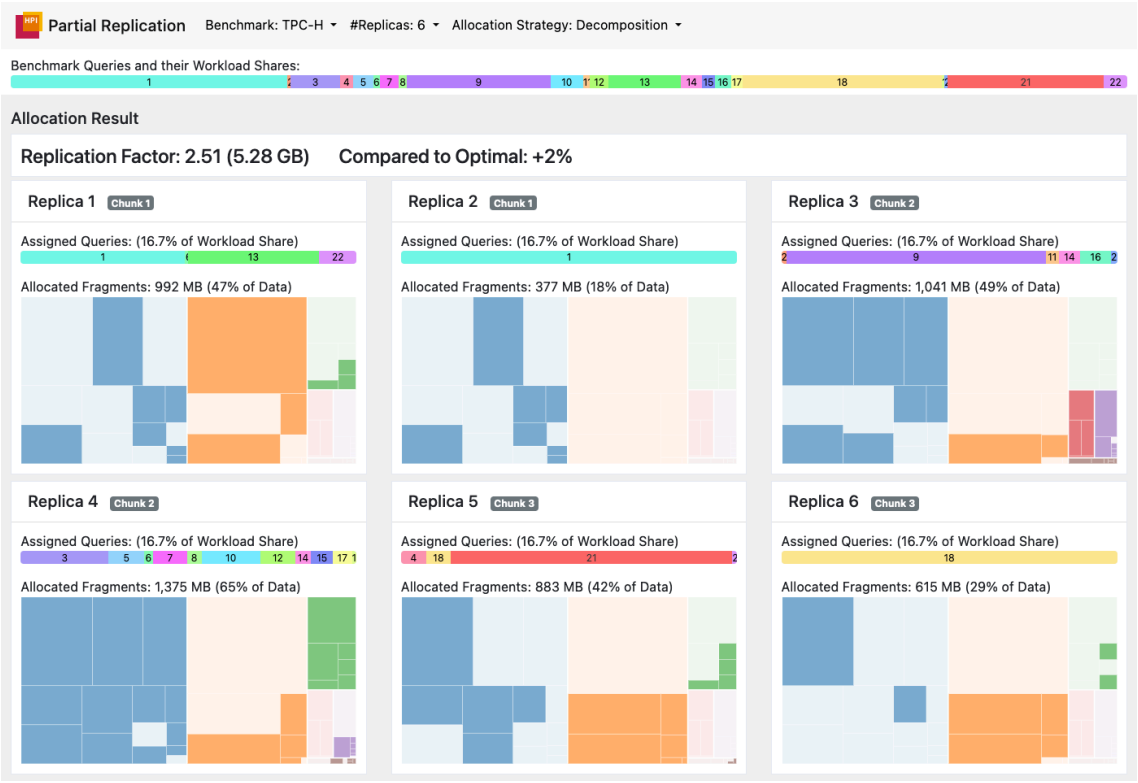


Figure 6.1.: The application screenshot shows an allocation of our decomposition heuristic for a cluster with six nodes running the TPC-H workload. [101]

tential for improvements. We, therefore, implemented an application to graphically show which fragments and queries are allocated to which nodes [101]. Figure 6.1 shows an application screenshot, which visualizes an allocation of our decomposition approach for the TPC-H benchmark using six replicas.

The workload shares of the individual queries are visualized below the navigation bar. We can derive that the execution costs for the queries 1, 9, 18, and 21 are the highest because their segment widths are the broadest. When hovering the mouse over a query segment, the required fragments for the query are shown. Below the navigation bar and the query list, the allocation result is visualized: (i) the replication factor and (ii) its comparison to an optimal allocation (if available). Further, six segments, one for each replica node, display detailed allocation information: (iii) assigned workload shares with regard to the queries and (iv) the allocated data fragments. The list of workload shares shows the assigned queries and visualizes to which extent they contribute to the replica load. For example, the TPC-H queries 1, 6, 13, and 22 are assigned to replica one. Thereby, query 1 accounts for the largest load on this replica. The allocated fragments are visualized as hierarchical map with nested rectangles. Rectangles with the same basic color represent fragments of the same table. Transparently colored fragments are not stored but show the memory

savings. The size of the rectangles corresponds to the fragment size. The 14 blue tiles model the 14 *accessed* columns of the TPC-H `lineitem` table, which accounts for the largest share of the data set.

End-to-End Evaluations

We also deployed allocations and executed workloads to obtain end-to-end results. In particular, we analyzed the query throughput of allocations. Besides the load limit/share L, the timing of queries influences the throughput of an allocation, in particular, whether each node can be used for query processing over time. Having batch processing or an even distribution of query loads concerning their assigned nodes, partially replicated database clusters performs optimally, i.e., in case of an evenly guaranteed load balancing by the model, the throughput of a cluster scales linearly with the number of nodes. In contrast, for our end-to-end evaluations, we use a set of query streams, simulating users. In this setting, single nodes may get overloaded temporarily while other nodes cannot be used for query processing, in particular in case of imbalanced workload shares.

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 $k = 1 + (s - 1) \bmod K$. For partial replication, queries are added to a queue of a node that 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 experiment with each setting for 620 s, executing more than 3 000 TPC-H and 3 300 TPC-DS queries for $K = 16$. We started measuring the query throughput after a 180 s warm-up phase.

Overall, the time of individual end-to-end experiments is mostly higher than the time to obtain numerical results. We, therefore, limit our end-to-end evaluation to selected, in our view essential, allocations, including solutions for the basic problem and situation with node failures, in which the load balancing can become skewed and the query throughput may drop.

Visualization of the Load Balancing. For end-to-end experiments, we log individual query executions and implemented an application to visualize the load balancing [103]: Our application allows users to retrace and evaluate the end-to-end performance of allocations in varying experiments, particularly situations in which single nodes are overloaded (e.g., due to a failed node), resulting in a lower query throughput. Figure 6.2 shows a screenshot of our application for a partially replicated database cluster running the TPC-H benchmark.

The visualization consists of three parts. First, the application's upper part shows the current replayed scenario. It comprises (from left to right) the benchmark, the

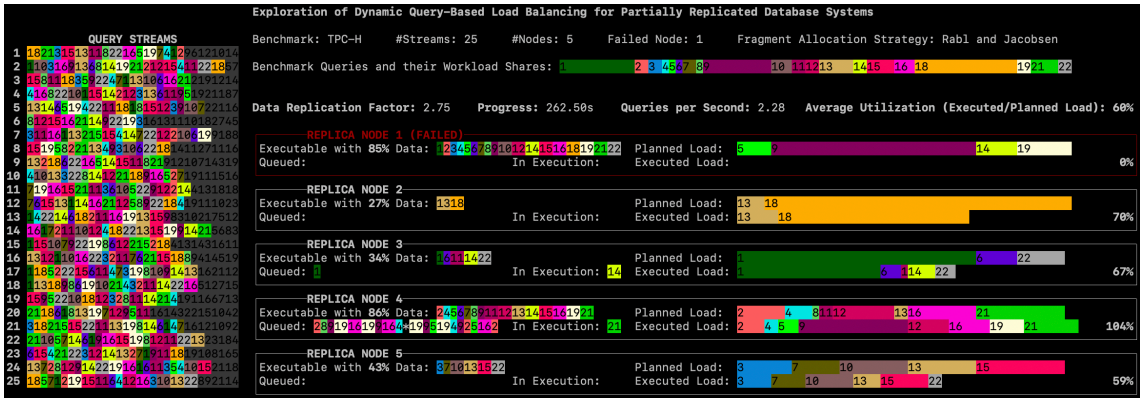


Figure 6.2.: The application screenshot shows a load-balancing status for running TPC-H queries, sent via 25 streams, in a five-node cluster, allocated using the greedy failure approach [181]. Node 1 has failed and cannot be used. Node 4 is overloaded, having a long query queue. Nodes 2, 3, and 5 are underutilized. [103]

number of concurrent query streams, the number of cluster nodes, (if existent) node failures, the used data allocation strategy, and (below) the workload shares of the benchmark queries. Each query class is visualized by a segment with a specific color tone, which is used throughout the application. The width of a segment corresponds to the workload share ($c_j \cdot f_j$) of the query.

Second, the left side of the application shows the workload as a set of 25 query streams. Each stream shows the order (from right to left) in which queries are submitted to the database cluster as a list of colored segments. Thereby, the color indicates the query class. Queries that have already been submitted to the cluster are shown as greyed out.

Third, the right side of the application shows allocation and load-balancing information of the replica nodes in individual boxes. Red boxes (see node 1) indicate failed nodes, which cannot be used for load-balancing. Each box consists of two lines. The first line summarizes static allocation information, i.e., the ratio of stored data, executable queries, and assigned/planned workload share per query. The second line of the box visualizes load-balancing information, i.e., currently queued queries and queries in execution, as well as a summary of the executed query load relative to the planned load considering the elapsed time. Thereby the executed load is calculated by summing up the product of query (execution) frequencies and planned costs. Because query execution costs may be lower than expected/modeled costs, the executed load can be (slightly) larger than 100% (see node 4).

The status summary above the node boxes shows the data replication factor (static), elapsed time, the executed queries per second since scenario start, and the average cluster *utilization* compared to a load execution with modeled query costs and without failures.

6.2. Evaluation of Models for the Basic Problem

In this section, we evaluate our allocation approaches for the basic problem, i.e., read-only workloads without any extensions. We first evaluate the optimal model in Section 6.2.1. Following, we present results of our decomposition-based heuristic (Section 6.2.2) and other approaches to lower the computation time (Section 6.2.3). Finally, we present end-to-end results and summarize our findings in Section 6.2.4.

6.2.1. Numerical Evaluation of the Optimal Solution

In the following, we evaluate the optimal model (see Section 5.1.1) for TPC-H, TPC-DS, and the accounting workload. We compare the solution quality (i.e., an allocation's normalized memory consumption or short the "replication factor" $\frac{W}{V}$ (see Section 3.1.2)) with allocations of the greedy state-of-the-art approach by Rabl and Jacobsen (see Section 4.3.1) and full replication: For different numbers of nodes K, Table 6.2 summarizes the calculation time, the optimal replication factor $\frac{W^*}{V}$, and memory savings compared to the greedy approach $\frac{W^*}{W^G}$ and full replication $\frac{W^*}{K \cdot V}$. All allocations achieve a perfect workload distribution $L^* = \frac{1}{K}$ (see Section 3.1.2).

The optimal replication factors $\frac{W^*}{V}$ (for $K > 1$) are 32 - 80% lower than the replication factors for full replication K, which demonstrates the potential memory (and thus cost) savings by using partial replication. With our setup (see Section 6.1.2), we could not calculate optimal allocations for TPC-DS with $K \geq 12$ and the accounting workload with $K \geq 6$ within 8 hours (see also [102, 181]). The reason is the increasing problem size, characterized by the number of variables and constraints. Further, the specific constraints, characterized by the parameter values (e.g., number of nodes K, fragment sizes a_i, query frequencies f_j, query costs c_j, and accessed fragments q_j) influence the problem complexity because they affect the solver's pruning capabilities. This is why, the calculation time of the optimal solution do not have to be monotonously increasing with the number of nodes K, e.g., the calculation time for the accounting workload with $K = 3$ (2369 s) is larger than for $K = 4$ (836 s).

Although the optimal solution approach is limited to smaller problems, it is extremely useful as it enables to analytically measure any heuristic's memory consumption (and calculation time). Compared to the greedy heuristic, the optimal solution reduces the memory consumption by up to 28%, 33%, and 43% for TPC-H, TPC-DS, and the accounting workload, respectively. Thereby, the solution quality of the greedy approach depends on the number of nodes K, e.g., $-29\% < \frac{W^*}{W^G} < -2\%$ for TPC-H, $-33\% < \frac{W^*}{W^G} < -16\%$ for TPC-DS, and $-43\% < \frac{W^*}{W^G} < -33\%$ for the accounting workload. Note, compared to the smaller TPC-H workload, the memory savings of optimal solutions are larger for the more complex TPC-DS and accounting workload. In contrast, the greedy approach is quick for all problem instances: The calculation time of our (unoptimized) Python implementation is < 1 s for TPC-H and DS, and only slightly higher for the accounting workload.

Table 6.2.: Optimal solutions of the **basic problem** for different workloads and cluster sizes K: calculation time, replication factor $\frac{W^*}{V}$, and relative memory savings compared to the greedy approach [181] $\frac{W^*}{W^G}$ and full replication $\frac{W^*}{K \cdot V}$.

K	time$_{W^*}$	$\frac{W^*}{V}$	$\frac{W^*}{W^G}$	$\frac{W^*}{K \cdot V}$
2	0.1 s	1.358	-13.9%	-32.1%
3	0.2 s	1.598	-28.3%	-46.7%
4	0.6 s	1.773	-7.6%	-55.7%
5	1.1 s	2.146	-19.3%	-57.1%
6	1.7 s	2.461	-12.1%	-59.0%
7	5.1 s	2.680	-12.3%	-61.7%
8	2.6 s	2.959	-13.9%	-63.0%
9	8.4 s	3.330	-12.0%	-63.0%
10	8.9 s	3.481	-14.3%	-65.2%
11	9.9 s	3.627	-2.3%	-67.0%
12	57.5 s	4.021	-9.0%	-66.5%
13	48.3 s	4.244	-11.1%	-67.4%
14	326.9 s	4.474	-11.4%	-68.0%
15	349.6 s	4.781	-6.8%	-68.1%
16	327.4 s	5.193	-9.6%	-67.5%

(a) TPC-H: N=61, Q=22, commercial database system, scale factor 10.

K	time$_{W^*}$	$\frac{W^*}{V}$	$\frac{W^*}{W^G}$	$\frac{W^*}{K \cdot V}$
2	0.8 s	1.142	-16.1%	-42.9%
3	2.7 s	1.278	-19.4%	-57.4%
4	41.8 s	1.441	-21.0%	-64.0%
5	87.0 s	1.559	-22.0%	-68.8%
6	131.8 s	1.691	-30.2%	-71.8%
7	227.7 s	1.749	-32.9%	-75.0%
8	343.9 s	1.852	-19.7%	-76.8%
9	2966.4 s	2.027	-27.9%	-77.5%
10	13227.5 s	2.115	-29.0%	-78.8%
11	20624.4 s	2.276	-25.8%	-79.3%

(b) TPC-DS: N=425, Q=95, commercial database system, scale factor 10.

K	time$_{W^*}$	$\frac{W^*}{V}$	$\frac{W^*}{W^G}$	$\frac{W^*}{K \cdot V}$
2	22.1 s	1.322	-33.9%	-33.9%
3	2368.8 s	1.774	-33.2%	-40.9%
4	835.8 s	2.104	-42.5%	-47.4%
5	38661.1 s	2.473	-36.3%	-50.5%

(c) Accounting workload: N=344, Q=4461, metadata.

Summary: *Our allocation problem can be solved optimally using ILP. However, for increasing problem sizes, the calculation times of optimal solutions may become too large for practical applicability. In these cases, we have to use heuristic approaches. Nevertheless, optimal solutions allow analyzing the quality of heuristics. In our case, the optimal solutions often have a more than 25% lower memory consumption than solutions of the greedy approach, particularly for the more complex TPC-DS and accounting workload.*

6.2.2. Numerical Evaluation of the Decomposition Heuristic

In this section, we illustrate the results of our decomposition heuristic (see Section 5.1.2) using TPC-H, TPC-DS, and the accounting workload. We compare the replication factors of our decomposition heuristic $\frac{W^D}{V}$ to solutions of the greedy state-of-the-art heuristic $\frac{W^G}{V}$ and to optimal solutions $\frac{W^*}{V}$ (if applicable). Again, all allocations achieve a perfect workload distribution $L^* = \frac{1}{K}$.

Naturally, the solution of our decomposition approach depends on the chunking, i.e., the number of decomposition levels as well as the number of chunks and chunk sizes per level. This is why, we state the chunking for each solution of our decomposition approach as a term: The number of summands indicates the number of chunks (per level). The summands themselves indicate the chunk sizes, i.e., the number of final nodes per chunk. The overall sum of all summands indicates the overall number of nodes K. Figure 6.3 visualizes exemplary chunkings for $K = 2 + 1 = 3$, $K = 2 + 2 = 4$, $K = 3 + 2 = 5$, $K = 3 + 3 = 6$, and $K = (4 + 4 + 4) + (4 + 4 + 4) = 24$. For symmetric chunkings with many levels and nodes, we state the chunking as (compact) product, e.g., $K = 2 + 2 = 2 \times 2 = 4$, $K = 3 + 3 = 2 \times 3 = 6$, $K = (4 + 4 + 4) + (4 + 4 + 4) = 2 \times (3 \times 4) = 24$, and $K = ((4 + 4) + (4 + 4)) + ((4 + 4) + (4 + 4)) = 2 \times (2 \times (2 \times 4)) = 32$.

Table 6.3, Table 6.4, and Table 6.5 summarize the results for TPC-H, TPC-DS, and the accounting workload, respectively, using different numbers of nodes

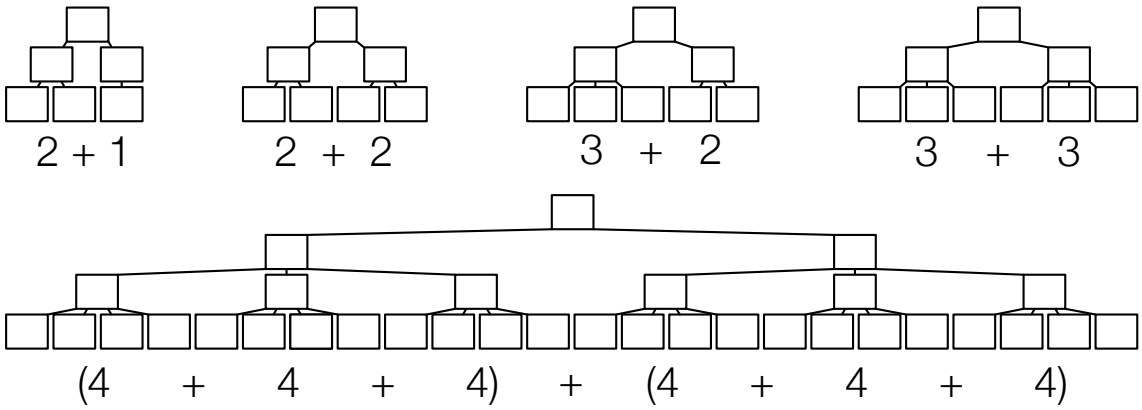


Figure 6.3.: Visualization of chunking terms.

Table 6.3.: Decomposition-based heuristic solutions of the **basic problem** for **TPC-H** with different cluster sizes K: chunk sizes of decomposition, calculation time, replication factor $\frac{W^D}{V}$, and relative memory consumption compared to the greedy approach [181] $\frac{W^D}{W^G}$ and optimal solution $\frac{W^D}{W^*}$.

K	chunking	time$_{W^D}$	$\frac{W^D}{V}$	$\frac{W^D}{W^G}$	$\frac{W^D}{W^*}$
3	2+1	0.1 s	1.598	-28.3%	+0.0%
4	2+2	0.1 s	1.842	-4.0%	+3.9%
5	3+2	0.2 s	2.220	-16.6%	+3.4%
6	2+2+2	0.3 s	2.511	-10.4%	+2.0%
7	4+3	0.3 s	2.720	-11.0%	+1.5%
8	2+2+2+2	0.4 s	2.968	-13.7%	+0.3%
9	3+3+3	0.3 s	3.339	-11.8%	+0.3%
10	3+3+2+2	1.0 s	3.592	-11.6%	+3.2%
11	3+3+3+2	1.0 s	3.627	-2.3%	+0.0%
12	6+6	0.6 s	4.061	-8.1%	+1.0%
13	7+6	0.8 s	4.299	-9.9%	+1.3%
14	5+5+4	0.5 s	4.515	-10.6%	+0.9%
15	5+5+5	0.9 s	4.791	-6.6%	+0.2%
16	4+4+4+4	0.6 s	5.255	-8.5%	+1.2%

$3 \leq K \leq 16$. Note, a decomposition with $K = 1 + 1 = 2$ nodes corresponds to the optimal solution and is therefore omitted. For TPC-DS (see Table 6.4), we additionally show the results of different chunkings (e.g., $K = 3 + 2 + 2 = 7$ vs. $K = 4 + 3 = 7$) and also larger number of nodes (i.e., $K = 24, 32, 48, 72, 100$).

We see that using the decomposition approach, we could calculate solutions for all $K \leq 16$, even for TPC-DS and the accounting workload, for which optimal solutions are limited to $K < 12$ and $K < 6$, respectively. Further, the calculation times are significantly lower than those of optimal solutions while the memory consumption W^D is *near-optimal* for the tractable optimal solutions W^*: Compared to the optimal data replication factors $\frac{W^*}{V}$ (see Table 6.2), the corresponding factors $\frac{W^D}{V}$ are only 0 - 5% larger. Compared to the replication factors of the greedy approach $\frac{W^G}{V}$, the memory consumption of our decomposition approach is *better for all investigated inputs*; for the more complex TPC-DS and accounting workload, the replication factors can often be reduced by more than 25% and 40%, respectively.

Via the number of levels and chunks chosen, we can control the problem complexity, which, in turn, makes it possible to even approach large problems. For TPC-DS, we could reduce the problem complexity such that we could calculate allocations for each $K \leq 100$ in less than 300 s. Partitionings with larger chunks require higher computation times but often lead to a lower memory consumption (see Table 6.4: 3+2+2 vs. 3+4 and 2×(2×(2×2)) vs. 4+4+4+4 vs. 8+8). Note, larger chunks do not guarantee better results if the intermediate results are different.

Table 6.4.: Decomposition-based heuristic solutions of the **basic problem** for **TPC-DS** with different cluster sizes K: chunk sizes of decomposition, calculation time, replication factor $\frac{W^D}{V}$, and relative memory consumption compared to the greedy approach [181] $\frac{W^D}{W^G}$ and optimal solution $\frac{W^D}{W^*}$.

K	chunking	time$_{W^D}$	$\frac{W^D}{V}$	$\frac{W^D}{W^G}$	$\frac{W^D}{W^*}$
3	2+1	2.1 s	1.278	-19.4%	+0.0%
4	2+2	1.6 s	1.445	-20.7%	+0.3%
5	3+2	6.2 s	1.574	-21.3%	+1.0%
6	3+3	6.1 s	1.695	-30.0%	+0.3%
7	3+2+2	8.9 s	1.828	-29.9%	+4.5%
7	4+3	40.1 s	1.796	-31.2%	+2.7%
8	3+3+2	6.0 s	1.908	-17.3%	+3.0%
8	4+4	28.7 s	1.879	-18.5%	+1.5%
9	3+3+3	5.1 s	2.046	-27.2%	+1.0%
10	4+3+3	40.1 s	2.160	-27.5%	+2.1%
11	4+4+3	31.1 s	2.337	-23.8%	+2.7%
12	4+4+4	51.7 s	2.463	-23.4%	-
13	4+3+3+3	34.8 s	2.599	-27.0%	-
13	5+4+4	103.8 s	2.559	-28.1%	-
14	4+4+3+3	27.2 s	2.664	-30.6%	-
15	4+4+4+3	68.2 s	2.828	-30.8%	-
15	5+5+5	105.1 s	2.781	-32.0%	-
16	2×(2×(2×2))	3.1 s	2.994	-32.3%	-
16	4+4+4+4	77.4 s	2.887	-34.8%	-
16	8+8	253.1 s	2.874	-35.1%	-
24	2×(3×4)	22.5 s	3.861	-23.5%	-
32	2×(2×(2×4))	78.7 s	4.728	-22.9%	-
48	2×(2×(3×4))	36.4 s	6.452	-14.4%	-
72	2×(3×(3×4))	21.5 s	9.059	-8.9%	-
100	2×(2×(5×5))	132.9 s	11.242	-11.8%	-

Table 6.5.: Decomposition-based heuristic solutions of the **basic problem** for the **accounting workload** with different cluster sizes K: chunk sizes of decomposition, calculation time, replication factor $\frac{W^D}{V}$, and relative memory consumption compared to the greedy approach [181] $\frac{W^D}{W^G}$ and optimal solution $\frac{W^D}{W^*}$.

K	chunking	time$_{W^D}$	$\frac{W^D}{V}$	$\frac{W^D}{W^G}$	$\frac{W^D}{W^*}$
3	2+1	41.8 s	1.781	-33.0%	+0.4%
4	2+2	43.0 s	2.131	-41.7%	+1.3%
5	2+2+1	413.2 s	2.538	-34.6%	+2.7%
6	3+3	582.9 s	2.911	-35.8%	-
7	3+2+2	2679.2 s	3.411	-38.4%	-
8	3+3+2	1436.1 s	3.531	-46.0%	-
9	3+3+3	331.5 s	4.010	-40.8%	-
10	4+3+3	4862.5 s	4.485	-42.2%	-
11	4+4+3	7628.0 s	4.756	-43.5%	-
12	4+4+4	3572.2 s	5.222	-44.5%	-
13	4+3+3+3	36015.1 s	5.540	-46.8%	-
14	4+4+3+3	14045.0 s	5.902	-45.2%	-
15	4+4+4+3	10234.3 s	6.537	-40.3%	-
16	4+4+4+4	7307.3 s	6.811	-37.4%	-

For the accounting workload, the calculation times remain high, i.e., multiple minutes or even hours for large cluster sizes K. We could have used a finer decomposition (as shown for TPC-DS). However, the solutions for small cluster sizes K indicate that the pure decomposition heuristic is not enough to obtain solutions within seconds for the large accounting workload.

Summary: *Our decomposition-based heuristic enables effective calculations for larger problem sizes. The chunk sizes can be chosen such that the computation times are small – even for the medium-sized TPC-DS workload. Compared to the greedy approach, solutions of the decomposition approach always reduced the memory consumption: depending on the workload complexity, often by more than 10% (TPC-H), 25% (TPC-DS), and 40% (accounting workload). The memory consumption was never more than 5% higher than optimal (for tractable optimal solution) while the calculation time is considerably faster. Still, using the decomposition approach alone does not enable solutions within seconds for the large-scale accounting workload.*

6.2.3. Numerical Evaluation of Approaches to Lower the Computation Time

In this section, we evaluate approaches to further lower the calculation time using the two larger TPC-DS and accounting workloads. Particularly, we apply optimality gaps, time limits, and query clustering to the optimal and decomposition approach.

Table 6.6.: Results of **optimality gap** approaches for different workloads and cluster sizes K: chunk sizes of decomposition, optimality gap, calculation time, replication factor $\frac{W^O}{V}$; calculation time relative to optimal $\frac{\text{time}W^O}{\text{time}W^*}$ and decomposition $\frac{\text{time}W^O}{\text{time}W^D}$ approach; memory consumption relative to optimal $\frac{W^O}{W^*}$, decomposition $\frac{W^O}{W^D}$, and greedy approach [181] $\frac{W^O}{W^G}$.

K	chunking	opt. gap	time$_{W^O}$	$\frac{W^O}{V}$	$\frac{\text{time}W^O}{\text{time}W^*}$	$\frac{W^O}{W^*}$	$\frac{\text{time}W^O}{\text{time}W^D}$	$\frac{W^O}{W^D}$	$\frac{W^O}{W^G}$
8	8	0.1	307.4 s	1.852	-10.6%	+0.0%	+5064.3%	-2.9%	-19.7%
8	8	0.2	58.6 s	1.859	-83.0%	+0.4%	+885.0%	-2.6%	-19.4%
8	8	0.3	39.7 s	1.878	-88.5%	+1.4%	+566.5%	-1.6%	-18.6%
8	8	0.4	16.4 s	1.957	-95.2%	+5.7%	+176.2%	+2.6%	-15.2%
8	8	0.5	9.2 s	1.972	-97.3%	+6.5%	+53.8%	+3.3%	-14.5%
16	4+4+4+4	0.1	15.7 s	2.912	-	-	-79.6%	+0.9%	-34.2%
16	4+4+4+4	0.2	6.8 s	2.982	-	-	-91.2%	+3.3%	-32.6%
16	4+4+4+4	0.3	3.3 s	3.204	-	-	-95.8%	+11.0%	-27.6%
16	4+4+4+4	0.4	2.2 s	3.240	-	-	-97.2%	+12.2%	-26.8%
16	4+4+4+4	0.5	0.9 s	3.490	-	-	-98.8%	+20.9%	-21.1%
16	16	0.4	1893.9 s	2.835	-	-	+2348.2%	-1.8%	-35.9%
16	16	0.5	102.4 s	2.847	-	-	+32.3%	-1.4%	-35.7%
16	16	0.6	24.0 s	2.880	-	-	-68.9%	-0.2%	-34.9%
16	16	0.7	5.7 s	3.139	-	-	-92.7%	+8.7%	-29.1%
16	16	0.8	3.6 s	3.545	-	-	-95.3%	+22.8%	-19.9%

(a) TPC-DS: N=425, Q=95, commercial database system, scale factor 10.

K	chunking	opt. gap	time$_{W^O}$	$\frac{W^O}{V}$	$\frac{\text{time}W^O}{\text{time}W^*}$	$\frac{W^O}{W^*}$	$\frac{\text{time}W^O}{\text{time}W^D}$	$\frac{W^O}{W^D}$	$\frac{W^O}{W^G}$
3	3	0.6	48.1 s	2.103	-98.0%	+18.5%	+15.1%	+18.1%	-20.8%
3	3	0.7	35.6 s	2.182	-98.5%	+23.0%	-14.7%	+22.5%	-17.9%
3	3	0.8	29.5 s	2.182	-98.8%	+23.0%	-29.3%	+22.5%	-17.9%
3	3	0.9	14.2 s	2.222	-99.4%	+25.2%	-66.1%	+24.8%	-16.4%
3	3	1.0	3.8 s	2.222	-99.8%	+25.2%	-91.0%	+24.8%	-16.4%
8	4+4	0.6	162.2 s	3.986	-	-	-88.7%	+12.9%	-39.0%
8	4+4	0.7	141.2 s	4.504	-	-	-90.2%	+27.6%	-31.1%
8	4+4	0.8	94.9 s	4.551	-	-	-93.4%	+28.9%	-30.4%
8	4+4	0.9	51.5 s	5.196	-	-	-96.4%	+47.2%	-20.5%
8	4+4	1.0	7.7 s	5.803	-	-	-99.5%	+64.4%	-11.2%
8	8	0.8	1002.2 s	7.413	-	-	-30.2%	+110.0%	+13.4%
8	8	0.9	160.9 s	7.413	-	-	-88.8%	+110.0%	+13.4%

(b) Accounting workload: N=344, Q=4461, metadata.

Optimality Gap

In the following, we evaluate the usage of different optimality gaps for TPC-DS and the accounting workload. We compare the potential time savings and memory sacrifices of solutions with a set optimality gap W^O against optimal solutions W^* (if possible) and solutions of the decomposition approach W^D. Further, we compare the memory consumption with optimality gaps W^O to the greedy approach W^G. Table 6.6a shows the results for the TPC-DS workload using the optimal base model with $K = 8, 16$ and the decomposition base model with $K = 4 + 4 + 4 + 4 = 16$ nodes. Analogously, Table 6.6b shows the results for the accounting workload using the optimal base model with $K = 3, 8$ and the decomposition base model with $K = 4 + 4 = 8$ nodes.

The results show that using optimality gaps can effectively and flexibly decrease the calculation time while sacrificing the memory consumption. When using the optimal model with a single ILP program (and usually also for the decomposition model), a larger optimality gap decreases the calculation time, but the memory consumption increases. Note, using a decomposition with multiple ILP programs, the optimality gap naturally influences the intermediate results, which are the input of subsequent ILP programs. Thus, better results with lower optimality gaps are not guaranteed.

For TPC-DS and specific optimality gaps (i.e., ($K = 8$, gap 0.5), ($K = 4 + 4 + 4 + 4 = 16$, gap 0.2), ($K = 16$, gap 0.7), we could reduce the calculation time below 10 s while the memory consumption is relatively close to optimal (less than +9%) compared to the base (optimal or decomposition) solution.

For the accounting workload and specific input settings (i.e., ($K = 3$, gap 0.6), ($K = 4 + 4 + 4 + 4 = 16$, gap 0.6)), we could also reduce the calculation time significantly (by more than −88%) while the increase in memory consumption is higher (less than +19%) compared to the base (optimal or decomposition) solution.

Further, (for the selected cluster sizes K) we could calculate solutions with a 19 - 35% lower memory consumption than the state-of-the-art greedy approach in under 60 s – even for the large accounting workload. However, for $K = 8$ and the optimal base model, we could not achieve a quick solution with a better memory consumption than the greedy approach, which demonstrates a limitation when not combining optimality gaps with other heuristics for larger problem sizes. Further, if particularly quick solutions (e.g., in less than 10 s) for mid or large-size problems are targeted, we find combining the decomposition approach with optimality gaps better than using optimality gaps on the optimal model (see Table 6.6a: $K = 4+4+4+4 = 16$, gap 0.2 vs. $K = 16$, gap 0.7).

Summary: *Using optimality gaps for our ILP-based optimal and decomposition model provides effective results that can (i) significantly reduce computation times and (ii) yield a near-optimal memory consumption. The approach effectively balances the computation time and solution quality from a practitioner's perspective.*

Time Limit

Next, we evaluate the usage of different time limits for TPC-DS and the accounting workload. As in the previous section, we use the optimal base model with $K = 8, 16$ and the decomposition base model with $K = 4 + 4 + 4 + 4 = 16$ nodes for TPC-DS. For the accounting workload, we use the optimal base model with $K = 3, 8$ and the decomposition base model with $K = 4 + 4$ nodes. Again, we compare the solutions of time limit approaches W^T against (pure) optimal W^* and (pure) decomposition-based W^D solutions as well as the solutions of the greedy approach. Table 6.7a and Table 6.7b show the results for TPC-DS and the accounting workload, respectively.

Table 6.7.: Results of **time limit** approaches for different workloads and cluster sizes K: chunk sizes of decomposition, time limit, calculation time, replication factor $\frac{W^T}{V}$; calculation time relative to optimal $\frac{\text{time}W^T}{\text{time}W^*}$ and decomposition $\frac{\text{time}W^T}{\text{time}W^D}$ approach; memory consumption relative to optimal $\frac{W^T}{W^*}$, decomposition $\frac{W^T}{W^D}$, and greedy approach [181] $\frac{W^T}{W^G}$.

K	chunking	time limit	time$_{W^T}$	$\frac{W^T}{V}$	$\frac{\text{time}W^T}{\text{time}W^*}$	$\frac{W^T}{W^*}$	$\frac{\text{time}W^T}{\text{time}W^D}$	$\frac{W^T}{W^D}$	$\frac{W^T}{W^G}$
8	8	300 s	300.2 s	1.852	-12.7%	-0.0%	+4944.3%	-2.9%	-19.7%
8	8	150 s	150.1 s	1.852	-56.4%	-0.0%	+2421.4%	-2.9%	-19.7%
8	8	60 s	60.1 s	1.852	-82.5%	-0.0%	+909.0%	-2.9%	-19.7%
8	8	30 s	30.1 s	1.934	-91.3%	+4.4%	+404.9%	+1.3%	-16.2%
8	8	10 s	10.0 s	2.205	-97.1%	+19.0%	+68.8%	+15.5%	-4.4%
16	4+4+4+4	(5 · 32 s =) 160 s	64.3 s	2.894	-	-	-16.9%	+0.2%	-34.6%
16	4+4+4+4	(5 · 16 s =) 80 s	32.4 s	2.984	-	-	-58.2%	+3.4%	-32.6%
16	4+4+4+4	(5 · 8 s =) 40 s	16.4 s	2.984	-	-	-78.8%	+3.4%	-32.6%
16	4+4+4+4	(5 · 4 s =) 20 s	8.3 s	3.007	-	-	-89.2%	+4.2%	-32.1%
16	4+4+4+4	(5 · 2 s =) 10 s	3.8 s	3.147	-	-	-95.1%	+9.0%	-28.9%
16	16	160 s	160.1 s	2.879	-	-	+106.9%	-0.3%	-34.9%
16	16	80 s	80.1 s	2.879	-	-	+3.5%	-0.3%	-34.9%
16	16	40 s	40.1 s	2.879	-	-	-48.2%	-0.3%	-34.9%
16	16	20 s	20.1 s	2.979	-	-	-74.1%	+3.2%	-32.7%
16	16	10 s	10.1 s	3.233	-	-	-87.0%	+12.0%	-26.9%

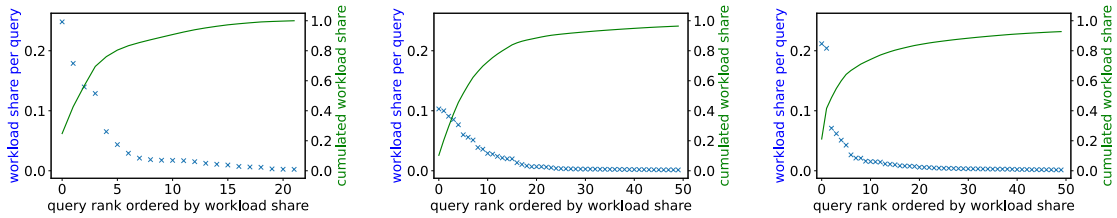
(a) TPC-DS: N=425, Q=95, commercial database system, scale factor 10.

K	chunking	time limit	time$_{W^T}$	$\frac{W^T}{V}$	$\frac{\text{time}W^T}{\text{time}W^*}$	$\frac{W^T}{W^*}$	$\frac{\text{time}W^T}{\text{time}W^D}$	$\frac{W^T}{W^D}$	$\frac{W^T}{W^G}$
3	3	600 s	600.3 s	1.859	-74.7%	+4.8%	+1337.4%	+4.4%	-30.0%
3	3	300 s	300.3 s	1.859	-87.3%	+4.8%	+619.1%	+4.4%	-30.0%
3	3	120 s	120.2 s	1.917	-94.9%	+8.1%	+187.9%	+7.7%	-27.8%
3	3	30 s	30.2 s	2.182	-98.7%	+23.0%	-27.7%	+22.5%	-17.9%
3	3	5 s	8.8 s	2.222	-99.6%	+25.2%	-78.9%	+24.8%	-16.4%
8	4+4	(3 · 300 s =) 900 s	333.9 s	3.817	-	-	-76.8%	+8.1%	-41.6%
8	4+4	(3 · 200 s =) 600 s	233.9 s	3.817	-	-	-83.7%	+8.1%	-41.6%
8	4+4	(3 · 100 s =) 300 s	134.0 s	3.924	-	-	-90.7%	+11.1%	-40.0%
8	4+4	(3 · 40 s =) 120 s	74.2 s	4.786	-	-	-94.8%	+35.5%	-26.8%
8	4+4	(3 · 20 s =) 60 s	52.1 s	4.786	-	-	-96.4%	+35.5%	-26.8%
8	4+4	(3 · 10 s =) 30 s	33.0 s	4.796	-	-	-97.7%	+35.9%	-26.6%
8	8	900 s	900.4 s	7.413	-	-	-37.3%	+110.0%	+13.4%
8	8	60 s	60.4 s	7.413	-	-	-95.8%	+110.0%	+13.4%

(b) Accounting workload: N=344, Q=4461, metadata.

Naturally, the results correspond with those of using an optimality gap. Using a decreasing time limit, we can flexibly sacrifice the solution quality (i.e., worsen the memory consumption) to obtain solutions more quickly. Compared to using optimality gaps, we can directly influence the calculation time of single ILP programs. For the decomposition approach, the total computation time is bounded from above by the chosen time limit multiplied by the number of splits performed.

For large-size problems (see accounting workload with $K = 8$ nodes), the usage of a time limit *alone* is not enough to obtain better solutions than the greedy approach. However, we can speed up single ILP instances significantly while barely increasing the memory consumption (see results using the optimal model: TPC-DS with $K = 8$ and accounting workload with $K = 3$, for which $\frac{\text{time}W^T}{\text{time}W^*} < -85\%$ while $\frac{W^T}{W^*} <$



(a) TPC-H. **(b)** TPC-DS. **(c)** Accounting workload.

Figure 6.4.: Distribution of query workload shares in decreasing order.

+5%). Combining time limits with our decomposition approach makes it possible to calculate solutions for the large-scale accounting workload in less than a minute while reducing the memory consumption compared to the greedy approach significantly (e.g., $\frac{W^T}{W^G} < -26\%$ for $K = 4 + 4$).

Summary: *The use of time limits for our ILP-based allocation approaches provides effective results similar to those of using optimality gaps. Instead of relaxing the memory consumption in the objective, it is possible to directly control the maximum calculation time. Naturally, both approaches can also be combined.*

Query Clustering

Finally, we evaluate our full clustering and partial clustering approach. Recap, using the full clustering approach, we assign as many as possible queries with a low workload share to a single node (e.g., the primary) without exceeding the maximum load $\frac{1}{K}$. For the partial clustering approach, we assign fewer queries so that the remaining load share can be used in the ILP program to assign memory-intensive queries flexibly.

We begin this section with a short analysis of the distribution of query loads for our three evaluation benchmarks TPC-H, TPC-DS, and the accounting workload. Following, we analyze the full and partial clustering approach for the two larger TPC-DS and accounting workloads.

Figure 6.4 visualizes the queries' individual workload shares and the resulting cumulated workload shares. We ordered the queries by their workload share $f_j \cdot q_j$. For TPC-H, we include all $Q = 22$ queries (see Figure 6.4a). For TPC-DS and the accounting workload, Figure 6.4b and 6.4c include only the top 50 queries with the highest workload shares for better clarity. The distribution is skewed for all three workloads: The top 50 queries account for more than 96% of the TPC-DS and more than 93% of the accounting workload. For TPC-H, the top half of the queries (i.e., $Q = 11$) accounts for 91%. We can exploit this skewness by fixedly assigning a large share of queries to a single node.

Table 6.8 and 6.9 show the results of our clustering approaches for TPC-DS and the accounting workload. The tables show the used number of fixedly assigned

Table 6.8.: Results of **query clustering** approaches for **TPC-DS** with different cluster sizes K: number of allocated $|Q^R|$ and fixed $|Q^F|$ queries, chunk decomposition, calculation time, replication factor $\frac{W^C}{V}$, calculation time relative to decomposition approach $\frac{\text{time}W^C}{\text{time}W^D}$, memory consumption relative to decomposition $\frac{W^C}{W^D}$, and greedy approach [181] $\frac{W^C}{W^G}$.

K	$\lvert Q^R\rvert + \lvert Q^F\rvert$	chunking	time$_{W^C}$	$\frac{W^C}{V}$	$\frac{\text{time}W^C}{\text{time}W^D}$	$\frac{W^C}{W^D}$	$\frac{W^C}{W^G}$
2	6+89	2	0.0 s	1.348	-95.8%	+18.0%	-1.0%
3	10+85	3	0.1 s	1.581	-97.6%	+23.6%	-0.3%
4	12+83	4	0.1 s	1.678	-91.3%	+16.1%	-8.0%
5	15+80	5	0.4 s	1.786	-93.9%	+13.5%	-10.7%
6	16+79	6	0.5 s	1.876	-91.5%	+10.7%	-22.5%
7	18+77	7	0.9 s	1.921	-97.8%	+7.0%	-26.4%
8	20+75	8	2.1 s	2.036	-65.0%	+6.7%	-11.7%
9	22+73	5+4	0.5 s	2.363	-89.4%	+15.5%	-15.9%
9	22+73	9	24.7 s	2.326	+384.0%	+13.7%	-17.2%
10	24+71	5+5	0.6 s	2.486	-98.6%	+15.1%	-16.6%
10	24+71	10	47.5 s	2.428	+18.5%	+12.4%	-18.6%
11	26+69	6+5	1.4 s	2.688	-95.5%	+15.0%	-12.4%
12	28+67	6+6	1.0 s	2.837	-98.1%	+15.2%	-11.8%
13	31+64	7+6	1.9 s	2.903	-94.6%	+11.7%	-18.4%
14	33+62	7+7	2.5 s	3.034	-90.9%	+13.9%	-20.9%
15	34+61	8+7	4.1 s	3.171	-94.1%	+12.1%	-22.5%
16	36+59	8+8	5.4 s	3.213	-93.0%	+11.3%	-27.4%

(a) Full clustering.

K	$\lvert Q^R\rvert + \lvert Q^F\rvert$	chunking	time$_{W^C}$	$\frac{W^C}{V}$	$\frac{\text{time}W^C}{\text{time}W^D}$	$\frac{W^C}{W^D}$	$\frac{W^C}{W^G}$
2	48+47	2	0.1 s	1.142	-92.9%	+0.0%	-16.1%
3	48+47	3	0.3 s	1.347	-87.9%	+5.4%	-15.1%
4	48+47	4	0.6 s	1.531	-61.9%	+5.9%	-16.0%
5	48+47	5	2.0 s	1.662	-67.7%	+5.6%	-16.9%
6	48+47	6	6.6 s	1.795	+8.4%	+5.9%	-25.9%
7	48+47	7	6.7 s	1.845	-25.1%	+0.9%	-29.3%
8	48+47	4+4	0.7 s	1.983	-88.9%	+3.9%	-14.0%
8	48+47	8	47.9 s	1.964	+703.9%	+2.9%	-14.9%
9	48+47	5+4	1.1 s	2.132	-78.6%	+4.2%	-24.2%
9	48+47	9	87.4 s	2.096	+1611.0%	+2.4%	-25.4%
10	48+47	5+5	1.3 s	2.300	-96.8%	+6.5%	-22.8%
11	48+47	6+5	3.6 s	2.470	-88.5%	+5.7%	-19.5%
12	48+47	6+6	2.0 s	2.585	-96.1%	+5.0%	-19.6%
13	48+47	7+6	4.0 s	2.709	-88.5%	+4.2%	-23.9%
14	48+47	7+7	5.5 s	2.809	-79.7%	+5.5%	-26.8%
15	48+47	8+7	7.9 s	2.941	-88.4%	+4.0%	-28.1%
16	48+47	6+5+5	1.7 s	3.037	-99.3%	+5.7%	-31.4%
16	48+47	8+8	18.0 s	3.048	-92.9%	+6.0%	-31.1%

(b) Partial clustering.

Table 6.9.: Results of **query clustering** approaches for the **accounting workload** with different cluster sizes K: number of allocated $|Q^R|$ and fixed $|Q^F|$ queries, chunk decomposition, calculation time, replication factor $\frac{W^C}{V}$, calculation time relative to decomposition approach $\frac{\text{time}W^C}{\text{time}W^D}$, memory consumption relative to decomposition $\frac{W^C}{W^D}$, and greedy approach [181] $\frac{W^C}{W^G}$.

K	$\lvert Q^R\rvert + \lvert Q^F\rvert$	chunking	time$_{W^C}$	$\frac{W^C}{V}$	$\frac{\text{time}W^C}{\text{time}W^D}$	$\frac{W^C}{W^D}$	$\frac{W^C}{W^G}$
2	4+4457	2	0.0 s	1.787	-99.9%	+35.1%	-10.7%
3	7+4454	3	0.2 s	2.087	-99.6%	+17.2%	-21.4%
4	12+4449	4	0.3 s	2.690	-99.4%	+26.2%	-26.4%
5	16+4445	5	0.7 s	3.098	-99.8%	+22.1%	-20.1%
6	20+4441	6	1.7 s	3.474	-99.7%	+19.4%	-23.4%
7	25+4436	7	3.0 s	3.722	-99.9%	+9.1%	-32.8%
8	29+4432	4+4	0.6 s	4.204	-100.0%	+19.1%	-35.7%
8	29+4432	8	22.1 s	4.198	-98.5%	+18.9%	-35.8%
9	33+4428	5+4	1.0 s	4.535	-99.7%	+13.1%	-33.0%
9	33+4428	9	1179.5 s	4.447	+255.8%	+10.9%	-34.3%
10	37+4424	5+5	1.6 s	4.844	-100.0%	+8.0%	-37.6%
11	40+4421	4+4+3	1.4 s	4.866	-100.0%	+2.3%	-42.2%
12	44+4417	4+4+4	1.4 s	5.945	-100.0%	+13.8%	-36.9%
13	47+4414	5+4+4	1.2 s	5.716	-100.0%	+3.2%	-45.1%
14	51+4410	5+5+4	1.6 s	6.103	-100.0%	+3.4%	-43.3%
15	55+4406	5+5+5	1.8 s	6.990	-100.0%	+6.9%	-36.2%
16	58+4403	6+5+5	2.0 s	7.066	-100.0%	+3.7%	-35.1%

(a) Full clustering.

K	$\lvert Q^R\rvert + \lvert Q^F\rvert$	chunking	time$_{W^C}$	$\frac{W^C}{V}$	$\frac{\text{time}W^C}{\text{time}W^D}$	$\frac{W^C}{W^D}$	$\frac{W^C}{W^G}$
2	100+4361	2	0.2 s	1.336	-99.2%	+1.1%	-33.2%
3	100+4361	3	0.5 s	1.891	-98.8%	+6.2%	-28.8%
4	100+4361	4	1.4 s	2.124	-96.7%	-0.3%	-41.9%
5	100+4361	5	3.2 s	2.492	-99.2%	-1.8%	-35.8%
6	100+4361	3+3	0.5 s	2.945	-99.9%	+1.2%	-35.1%
6	100+4361	6	64.2 s	2.932	-89.0%	+0.7%	-35.4%
7	100+4361	4+3	1.2 s	3.291	-100.0%	-3.5%	-40.6%
8	100+4361	4+4	1.2 s	3.534	-99.9%	+0.1%	-45.9%
9	100+4361	3+3+3	1.1 s	4.435	-99.7%	+10.6%	-34.5%
9	100+4361	5+4	11.5 s	4.261	-96.5%	+6.3%	-37.1%
10	100+4361	5+5	3.7 s	4.638	-99.9%	+3.4%	-40.2%
11	100+4361	4+4+3	1.2 s	4.867	-100.0%	+2.3%	-42.2%
12	100+4361	4+4+4	1.6 s	5.513	-100.0%	+5.6%	-41.4%
13	100+4361	5+4+4	1.4 s	5.641	-100.0%	+1.8%	-45.8%
14	100+4361	5+5+4	1.6 s	5.945	-100.0%	+0.7%	-44.8%
15	100+4361	5+5+5	3.3 s	6.785	-100.0%	+3.8%	-38.0%
16	100+4361	6+5+5	3.4 s	6.804	-100.0%	-0.1%	-37.5%

(b) Partial clustering.

queries $|Q^F|$ (and the number of remaining queries to assign $|Q^R| = Q - |Q^F|$), the chunk decomposition, the calculation time of the clustering heuristic, and the solutions' (normalized) memory consumption $\frac{W^C}{V}$. Further, we compare the calculation time (see $\frac{\text{time}W^C}{\text{time}W^D}$) and memory consumption (see $\frac{W^C}{W^D}$) against our decomposition-based approach. Finally, the tables include the memory savings compared to the greedy approach $\frac{W^C}{W^G}$. We summarize our results as follows.

Even for the TPC-DS workload with $Q = 95$ queries, the majority of queries can be assigned to a single node: Using the full clustering approach, $|Q^F| = 59$ (for $K = 16$) to $|Q^F| = 89$ (for $K = 2$) queries are fixedly assigned without exceeding the load limit $\frac{1}{K}$. As a result, the ILP model's complexity and, thus, calculation time can be reduced significantly – even compared to the relatively quick decomposition approach. As a consequence, we achieve computation times below 10 s for cluster sizes $K \leq 8$ without using a decomposition approach for the $|Q^R|$ remaining queries. For a larger number of nodes $K > 8$, we can combine the full clustering and decomposition approach to obtain solutions quickly. However, the memory consumption increases by 6 - 24% compared to the decomposition approach, particularly for small $K \leq 3$. Still, we achieve a 0 - 28% lower memory consumption than the greedy approach for all $K = 2, ..., 16$.

The partial clustering heuristic is more flexible than full clustering and enables better results. Naturally, the number of fixed queries can be chosen so that from one extreme, we achieve an optimal solution (using $|Q^F| = 0$) to the other extreme, for which we use the full clustering heuristic. With a decreasing number of fixed queries $|Q^F|$, the memory consumption improves because the remaining allocation is more flexible. However, the runtime is increasing due to the larger remaining problem size. In Table 6.8b, we use $|Q^F| = 47$ (i.e., clustering half of the queries) for all cluster sizes $2 \leq K \leq 16$, which we found a good tradeoff with low computation times and a low memory consumption. Overall, we achieve better allocations with a lower memory consumption than the full clustering approach – still using less than 10 s if needed. Because of the higher complexity compared to full clustering, we have to combine the partial clustering with our decomposition approach already for cluster sizes $K > 7$ and use more chunks for $K = 16 = 6 + 5 + 5$ (vs. $K = 8 + 8$) to achieve calculation times below 10 s. The memory consumption of the partial clustering approach is clearly better than for full clustering. Further, the memory consumption is close to the decomposition approach (i.e., $\frac{W^C}{W^D} < +7\%$) and 14 - 32% lower compared to the greedy approach.

The results for the accounting workload are overall similar, but the advantages of clustering are larger than for the smaller TPC-DS workload. For the partial clustering heuristic, we use $|Q^F| = 4361$ clustered queries, resulting in a good tradeoff with low computation times and a low memory consumption. As a result, we achieve solutions for all cluster sizes $K \leq 16$ in under 10 s, reducing the calculation time by over 96% compared to the decomposition approach. Even for small cluster sizes $2 \leq K \leq 6$, the full clustering approach reduces the memory consumption compared to the greedy approach by 10 - 27%. Nevertheless, we find the partial clustering

approach (as for TPC-DS) superior over full clustering: Compared to the decomposition approach, partial clustering requires only $< +7\%$ more memory (for selected chunkings, i.e., $K = 6 = 3 + 3$ and $K = 9 = 5 + 4$) while it calculates solutions in 12 s or less – recap, the decomposition approach takes over an hour for $K > 9$ (see Table 6.5). Note, we can even achieve a lower memory consumption than the decomposition approach (e.g., $W^C < W^D$ for $K = 4, 5, 7, 16$) due to different intermediate results or using no decomposition at all (compare chunking against Table 6.5).

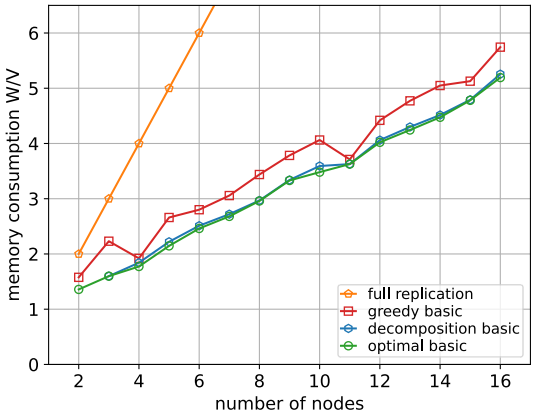
Summary: *If the queries' workload shares are skewed (which is typical for real-world workloads), it is reasonable to exploit the property that many queries can be clustered and assigned to a single node. The remaining set of queries is smaller and can, thus, be allocated considerably faster. For small problem sizes or when higher calculation times are admissible, using a pure decomposition approach provides slightly better results and is, thus, still favorable. Overall, all techniques for lowering the computation time (i.e., the decomposition approach, solver relaxations, and query clustering) are compatible and can be flexibly combined.*

6.2.4. End-to-End Evaluation and Summary

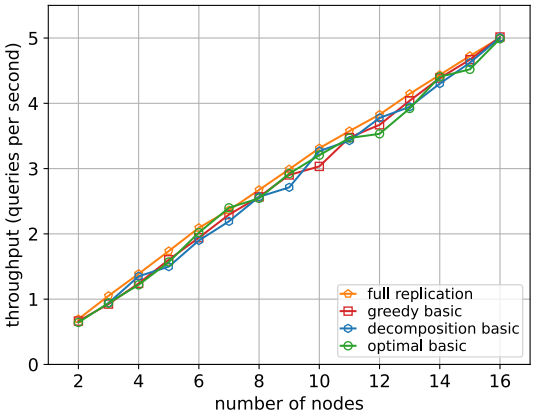
In this section, we summarize our results for the basic (read-only) problem and study the performance of allocations against other approaches in end-to-end settings. Figure 6.5a, 6.5c, and 6.5e visualize the memory consumption of different ILP-based approaches, the greedy state-of-the-art approach, and full replication for TPC-H, TPC-DS, and the accounting workload, respectively. Figure 6.5b and 6.5d show the end-to-end query throughput of deployed allocations, running TPC-H and TPC-DS workloads in our cluster of columnar, in-memory database instances. Recap, for the accounting workload, we had only metadata. This is why, Figure 6.5f shows a theoretically possible linear throughput scaling for all approaches.

For TPC-H and TPC-DS, all allocation approaches provide the *same throughput scaling* (see Figure 6.5b, 6.5d). Thereby, the replication factors of the decomposition heuristic are close to optimal, whereas the memory consumption of the greedy heuristic is comparably far off. Recap, particularly for the more complex TPC-DS and accounting workload, the decomposition approach requires up to 35% (see Table 6.4) and 47% (see Table 6.5) less data.

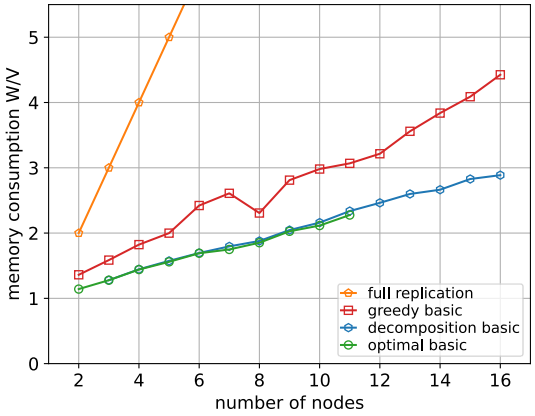
The query throughput of partial allocation algorithms scales linearly. Nevertheless, for the TPC-DS benchmark with more queries, full replication has a slightly higher throughput for large cluster sizes. With increasing cluster sizes, partial replicas are more and more specialized, having fewer fragments and executable queries. Full replication is more flexible because each replica can execute every query. In contrast, for partial replication, a few replicas may possibly have no query available while others are temporarily overloaded (see Section 6.1.3). Analyzing query execution times and query timings, we made the following additional observations: Partial replication can reduce query execution times. Rabl and Jacobsen explain



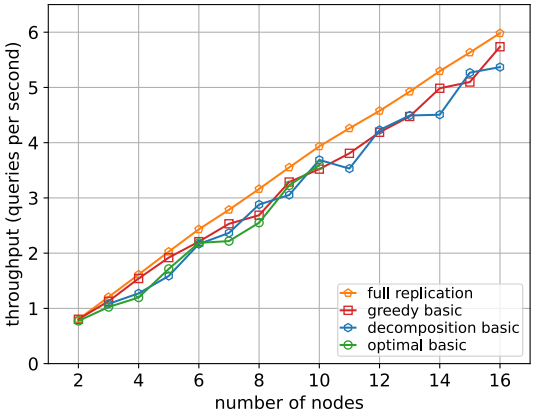
(a) TPC-H: memory consumption.



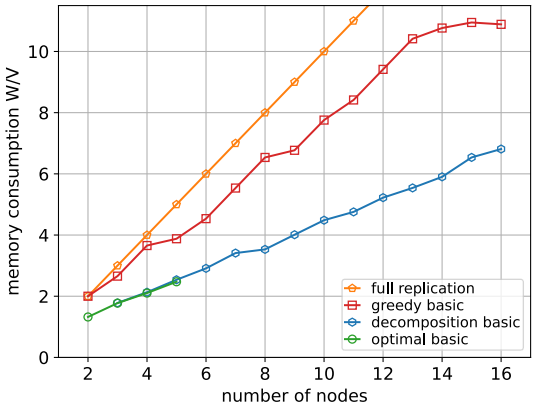
(b) TPC-H: end-to-end throughput.



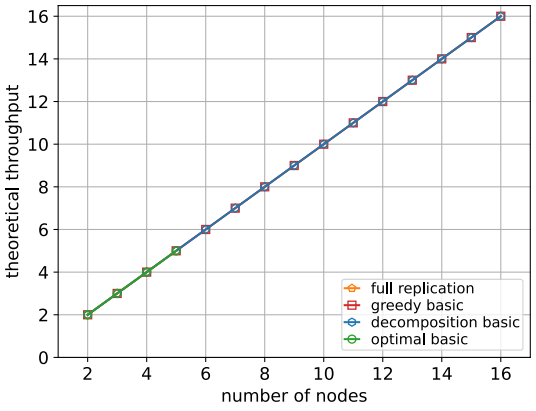
(c) TPC-DS: memory consumption.



(d) TPC-DS: end-to-end throughput.



(e) Accounting: memory consumption.



(f) Accounting: numerical throughput.

Figure 6.5.: Evaluation of allocations for the **basic problem** for TPC-H, TPC-DS, and the accounting workload with 2 - 16 nodes: memory consumption and throughput for full replication, the greedy basic approach, the decomposition approach, and optimal solutions.

this speed-up compared to full replication with better caching because nodes process only subsets of all queries [181]. Finally, we measured variances of execution times for queries of the same class, e.g., caused by different query template parameters. Variances can be handled by the dispatcher, which distributes queries to the replicas dynamically with regard to the estimated costs of currently processed queries and those in the queues.

Summary: *The end-to-end evaluations correspond to the numerical results. Our ILP-based partial allocations can scale the throughput (almost) linearly while requiring less memory than the state-of-the-art greedy approach. Overall, the memory consumption of our heuristics is close to optimal (if comparable) while we could calculate solutions for all workloads and cluster sizes using only a few seconds.*

6.3. Evaluation of Model Extensions

In this section, we evaluate our model extensions that consider node failures (Section 6.3.1), workload uncertainty (Section 6.3.2), data modifications (Section 6.3.3), and reallocation costs (Section 6.3.4).

6.3.1. Node Failures

This section presents our evaluation of allocation approaches that consider node failures. For the evaluation, we use TPC-H, TPC-DS, and the accounting workload. We evaluate cases with $3 \leq K \leq 16$ nodes in which single nodes fail. We want to calculate allocations that can balance the workload evenly without a node failure and for each single-node failure. For $K = 2$, the solution is trivial because both nodes have to store the same fragments. We compare our optimal and heuristic solutions (see Section 5.3.1) against the greedy extension for node failures and the chaining approach (see Section 4.3.1) that uses a basic greedy solution as basis allocation.

Recap, solutions of the greedy failure approach do not guarantee an even load balancing in failure cases. We can use our ILP model (5.12) - (5.19) with a fixed fragment allocation (that is calculated with the greedy failure approach) for calculating the maximum load that a single node has to take in failure cases, hereinafter called "worst-case workload share". Solutions of the chaining approach achieve a perfect workload distribution without a failure and for each single-node failure case.

In the following, we first numerically evaluate the optimal ILP model that considers node failures. Then, we present results of our heuristic ILP-based approaches. We conclude this section with end-to-end evaluations and a summary.

Numerical Evaluation of the Optimal Solution

For different numbers of nodes K, Table 6.10 summarizes the calculation time, the optimal replication factor $\frac{W^{F*}}{V}$, and the optimal worst-case workload share

$L_{max}^{(-)} = \frac{1}{K-1}$. Further, the table compares the optimal solutions W^{F*} to solutions of the greedy failure approach W^{GF} and chaining approach W^{CF} with regard to the memory consumption. We also compare the optimal worst-case workload shares $L_{max}^{(-)}$ with the worst-case workload shares of the greedy failure approach $L_{max}^{GF(-)}$. (The worst-case workload shares of the chaining approach $L_{max}^{CF(-)}$ are optimal, i.e., $L_{max}^{(-)} = L_{max}^{CF(-)}$.) For TPC-H with $K = 4, 7, 11$ nodes, we also show the results of the optimal robust model for the worst-case workload share of the greedy failure approach, i.e., with $L_{max}^{(-)} := L_{max}^{RG(-)}$ (see Table 6.10a, bottom part).

We summarize the main results: The calculation time for optimal solutions that consider single-node failures is higher compared to basic solutions (see Table 6.2), which do not consider node failures. The reason for the higher complexity is that suitable workload distributions for each failure case have to be found. As a result, optimal allocations for failure cases were only tractable for TPC-H with $K < 13$, for TPC-DS with $K < 8$, and for the accounting workload with $K < 4$, i.e., we could not achieve optimal solutions for larger K with our setup and a set time limit of 8 hours. Naturally, the necessary data replication factors are also higher compared to the basic model (see Table 6.2) because usually additional fragments are required to ensure robustness against node failures.

Further, the results of Table 6.10 show that our optimal failure solution outperforms the greedy failure and chaining approach with regard to the memory consumption and worst-case workload share: Compared to the greedy failure approach, we reduce the worst-case workload share in failure cases by up to 44% (see TPC-H, $K = 11$) while our memory consumption is similar ($-8\% \leq \frac{W^{F*}}{W^{GF}} \leq +8\%$). Note, the greedy failure approach can only have a lower memory consumption than the optimal failure approach (see TPC-H, $K = 4, 7, 11$) if its worst-case workload share $L_{max}^{GF(-)}$ is *worse* than the optimal share $L_{max}^{(-)} = \frac{1}{K-1}$. Using the optimal failure model with the worst-case workload share of the greedy failure approach (i.e., $L_{max}^{(-)} := L_{max}^{GF(-)}$), we can always calculate a solution with the same (see TPC-H, $K = 4$, $L_{max}^{(-)} = 0.481$) or a lower (see TPC-H, $K = 7$, $L_{max}^{(-)} = 0.248$ and $K = 11$, $L_{max}^{(-)} = 0.179$) memory consumption than the greedy failure approach. The possibility to *directly control the worst-case workload share* is an advantage of ILP-based approaches because it is impossible using the greedy failure or chaining approach.

For TPC-DS, we require up to 17% (see $K = 7$) less memory than the greedy approach while in addition reducing the worst-case workload share in failure cases by 17%. Our replication factors and worst-case workload shares are lower than those of the greedy approach for all calculated TPC-DS solutions. For the accounting workload, we could only calculate the optimal failure solution for $K = 3$, for which we require 13% less memory than the greedy approach having the same (optimal) worst-case workload share in failure cases.

Compared to the chaining approach, the optimal failure solutions lower the memory consumption for all workloads more substantially (10% - 33% less) while providing the same worst-case workload shares.

Table 6.10.: Optimal solutions of allocation approaches that consider **node failures** for different workloads and cluster sizes K: calculation time, replication factor $\frac{W^{F*}}{V}$, (optimal) worst-cast workload share for all K single-node failure cases, relative memory consumption $\frac{W^{F*}}{W^{GF}}$ and relative worst-cast workload share improvement $\frac{L_{max}^{(-)}}{L_{max}^{GF(-)}}$ compared to the greedy failure approach [181], and relative memory consumption $\frac{W^{F*}}{W^{CF}}$ compared to the chaining approach [108]. "L" indicates a solution with a suboptimal worst-cast workload share.

K	time$_{W^{F*}}$	$\frac{W^{F*}}{V}$	$L_{max}^{(-)}$	$\frac{W^{F*}}{W^{GF}}$	$\frac{L_{max}^{(-)}}{L_{max}^{GF(-)}}$	$\frac{W^{F*}}{W^{CF}}$
3	0.3 s	2.358	0.500	-7.7%	-0.0%	-12.3%
4	0.9 s	2.554	0.333	+6.5%	-30.7%	-19.7%
5	4.5 s	2.773	0.250	-6.4%	-21.8%	-24.4%
6	50.8 s	3.074	0.200	-0.9%	-30.4%	-23.7%
7	182.5 s	3.460	0.167	+2.5%	-32.8%	-21.5%
8	478.0 s	3.680	0.143	-1.5%	-42.4%	-23.1%
9	1005.0 s	3.959	0.125	-3.0%	-30.1%	-25.8%
10	4369.0 s	4.165	0.111	-6.0%	-37.9%	-27.4%
11	9854.0 s	4.441	0.100	+8.0%	-44.1%	-15.2%
12	14433.0 s	4.617	0.091	-2.4%	-34.9%	-21.6%
4	L 0.6 s	L 2.398	L 0.481	L -0.0%	L -0.0%	L -24.6%
7	L 22.6 s	L 3.190	L 0.248	L -5.5%	L -0.0%	L -27.7%
11	L 851.0 s	L 4.066	L 0.179	L -1.2%	L -0.0%	L -22.4%

(a) TPC-H: N=61, Q=22, commercial database system, scale factor 10.

K	time$_{W^{F*}}$	$\frac{W^{F*}}{V}$	$L_{max}^{(-)}$	$\frac{W^{F*}}{W^{GF}}$	$\frac{L_{max}^{(-)}}{L_{max}^{GF(-)}}$	$\frac{W^{F*}}{W^{CF}}$
3	5.5 s	2.142	0.500	-5.2%	-4.2%	-10.3%
4	23.8 s	2.278	0.333	-4.5%	-22.6%	-17.9%
5	865.9 s	2.441	0.250	-0.8%	-26.4%	-18.4%
6	7213.7 s	2.559	0.200	-10.8%	-25.5%	-28.8%
7	29484.0 s	2.690	0.167	-16.9%	-16.8%	-32.7%

(b) TPC-DS: N=425, Q=95, commercial database system, scale factor 10.

K	time$_{W^{F*}}$	$\frac{W^{F*}}{V}$	$L_{max}^{(-)}$	$\frac{W^{F*}}{W^{GF}}$	$\frac{L_{max}^{(-)}}{L_{max}^{GF(-)}}$	$\frac{W^{F*}}{W^{CF}}$
3	169.8 s	2.322	0.500	-12.6%	-0.0%	-22.6%

(c) Accounting workload: N=344, Q=4461, metadata.

Overall, even though optimal solutions that consider node failures are only tractable for small problem instances, our derived solutions show the potential to improve the quality of different heuristic approaches.

Summary: *Our ILP model can be extended to also enable an even load balancing in failure cases. Compared to state-of-the-art approaches, (our calculated) optimal solutions reduce the worst-case workload shares in failure cases by up to 44% and require up to 33% less memory. As optimal solutions can only be calculated in a reasonable amount of time for smaller problem instances, memory-efficient and robust heuristic solutions are required.*

Numerical Evaluation of ILP-Based Heuristics

In the following, we evaluate our ILP-based heuristic approaches that consider node failures for TPC-H, TPC-DS, and the accounting workload. For different numbers of nodes K, we summarize the used decomposition's chunking, the calculation time, the replication factor $\frac{W^F}{V}$, and the corresponding worst-case workload share $L_{max}^{(-)} = \frac{1}{K-1}$. Further, we compare our solutions W^F to solutions of the greedy failure approach W^{GF} and chaining approach W^{CF} with regard to the memory consumption and worst-case workload shares of nodes in failure cases. Solutions of the chaining approach achieve a perfect workload distribution in failure cases, i.e., $\frac{L_{max}^{(-)}}{L_{max}^{CF(-)}} = 1$.

For TPC-H, Table 6.11 shows the results for the three-step and two-step approach. Our solutions demonstrate that (in contrast to the optimal failure model) even large problems can be solved quickly: the maximum calculation time of the provided two-step approaches is 16 s for all $K \leq 16$. We achieved these calculation times without query clustering or solver relaxations (i.e., time limits and optimality gaps).

For TPC-H, the calculation time of our two-step approach (see Table 6.11b) is quick while having a lower memory consumption than the three-step approach (see Table 6.11a). Hence, we focus on the results of our two-step approach when comparing to the state-of-the-art approaches. We find that our worst-case workload limits $L_{max}^{(-)}$ are *clearly better* (up to 45% for $K = 14$) compared to the solutions of the greedy failure approach, although our *two-step* approach requires similar or even less memory ($-7\% < \frac{W^F}{W^{GF}} < +9\%$). Actually, we were able to always provide solutions with the same or lower memory consumption than the greedy approach when using their worst-case workload share as input, i.e., $L_{max}^{(-)} := L_{max}^{GF(-)}$ (see Table 6.11b, bottom part): Recap, the workload limit $L_{max}^{(-)}$ of our approaches coincides with the optimal lower bound $L^{(-)*} = \frac{1}{K-1}$. Instead, the limit $L_{max}^{GF(-)}$ of the greedy approach is significantly worse ($\frac{L_{max}^{(-)}}{L_{max}^{GR(-)}} < -20\%$) for most K (except $K = 3$) and can be close to the worst case $2 \cdot L^*$ (see TPC-H, $K = 14$, for which $L_{max}^{GF(-)} = 0.140$ and $2 \cdot L^* = \frac{2}{K} = 0.143$), which reflects the case in which one node has to (i) additionally take the entire workload of the failure node ($\frac{1}{K}$) and (ii) cannot pass some of its regular workload ($\frac{1}{K}$) to other nodes. Naturally, such scenarios, in which the throughput is cut in half, are critical in practice and should be avoided.

Compared to the chaining approach, our two-step approach requires less memory for all $3 \leq K \leq 16$ ($-27\% \leq \frac{W^{RD}}{W^{RC}} \leq -11\%$) while providing the same worst-case

Table 6.11.: Decomposition-based heuristic solutions that consider **node failures** for **TPC-H** with different cluster sizes K: chunk decomposition, calculation time, replication factor $\frac{W^F}{V}$, (optimal) worst-cast workload share for all K single-node failure cases, relative memory consumption $\frac{W^F}{W^{GF}}$ and relative worst-cast workload share improvement $\frac{L_{max}^{(-)}}{L_{max}^{GF(-)}}$ compared to the greedy failure approach [181], and relative memory consumption $\frac{W^F}{W^{CF}}$ compared to the chaining approach [108]. "L" indicates a solution with a suboptimal worst-cast workload share.

K	chunking	time$_{W^F}$	$\frac{W^F}{V}$	$L_{max}^{(-)}$	$\frac{W^F}{W^{GF}}$	$\frac{L_{max}^{(-)}}{L_{max}^{GF(-)}}$	$\frac{W^F}{W^{CF}}$
3	2+1	0.1 s	2.443	0.500	-4.3%	-0.0%	-9.2%
4	2+2	0.1 s	2.851	0.333	+18.9%	-30.7%	-10.4%
5	3+2	0.3 s	3.113	0.250	+5.1%	-21.8%	-15.2%
6	3+3	0.3 s	3.405	0.200	+9.8%	-30.4%	-15.4%
7	4+3	1.1 s	3.809	0.167	+12.8%	-32.8%	-13.6%
8	4+4	1.5 s	4.011	0.143	+7.4%	-42.4%	-16.1%
9	5+4	1.7 s	4.348	0.125	+6.6%	-30.1%	-18.5%
10	5+5	2.2 s	4.452	0.111	+0.5%	-37.9%	-22.4%
11	6+5	2.0 s	4.809	0.100	+16.9%	-44.1%	-8.2%
12	6+6	4.9 s	5.035	0.091	+6.4%	-34.9%	-14.5%
13	7+6	7.0 s	5.463	0.083	+7.7%	-40.3%	-11.8%
14	7+7	62.7 s	5.496	0.077	+2.9%	-44.9%	-18.4%
15	8+7	12.2 s	5.826	0.071	+6.0%	-44.6%	-11.7%
16	8+8	282.9 s	6.186	0.067	+4.8%	-25.5%	-15.5%

(a) Three-step approach.

K	chunking	time$_{W^F}$	$\frac{W^F}{V}$	$L_{max}^{(-)}$	$\frac{W^F}{W^{GF}}$	$\frac{L_{max}^{(-)}}{L_{max}^{GF(-)}}$	$\frac{W^F}{W^{CF}}$
3	3	0.2 s	2.376	0.500	-6.9%	-0.0%	-11.6%
4	4	0.8 s	2.564	0.333	+6.9%	-30.7%	-19.4%
5	5	1.2 s	2.804	0.250	-5.4%	-21.8%	-23.6%
6	6	1.8 s	3.089	0.200	-0.4%	-30.4%	-23.3%
7	7	5.8 s	3.479	0.167	+3.0%	-32.8%	-21.1%
8	8	3.9 s	3.680	0.143	-1.5%	-42.4%	-23.1%
9	9	9.1 s	4.068	0.125	-0.3%	-30.1%	-23.7%
10	10	10.3 s	4.186	0.111	-5.5%	-37.9%	-27.0%
11	3+3+3+2	9.9 s	4.481	0.100	+8.9%	-44.1%	-14.5%
12	6+6	8.1 s	4.638	0.091	-2.0%	-34.9%	-21.2%
13	7+6	11.5 s	5.062	0.083	-0.2%	-40.3%	-18.2%
14	5+5+4	16.2 s	5.283	0.077	-1.0%	-44.9%	-21.5%
15	5+5+5	12.5 s	5.535	0.071	+0.7%	-44.6%	-16.1%
16	4+4+4+4	7.6 s	5.757	0.067	-2.4%	-25.5%	-21.3%
4	4	L 0.6 s	L 2.398	L 0.481	L -0.0%	L -0.0%	L -24.6%
7	7	L 5.3 s	L 3.214	L 0.248	L -4.8%	L -0.0%	L -27.1%
11	3+3+3+2	L 1.1 s	L 4.076	L 0.179	L -0.9%	L -0.0%	L -22.2%
15	5+5+5	L 1.2 s	L 5.226	L 0.129	L -4.9%	L -0.0%	L -20.8%

(b) Two-step approach.

Table 6.12.: Decomposition-based heuristic solutions that consider **node failures** for **TPC-DS** with different cluster sizes K: chunk decomposition, calculation time, replication factor $\frac{W^F}{V}$, (optimal) worst-cast workload share for all K single-node failure cases, relative memory consumption $\frac{W^F}{W^{GF}}$ and relative worst-cast workload share improvement $\frac{L_{max}^{(-)}}{L_{max}^{GF(-)}}$ compared to the greedy failure approach [181], and relative memory consumption $\frac{W^F}{W^{CF}}$ compared to the chaining approach [108]. "L" indicates a solution with a suboptimal worst-cast workload share.

K	chunking	time$_{W^F}$	$\frac{W^F}{V}$	$L_{max}^{(-)}$	$\frac{W^F}{W^{GF}}$	$\frac{L_{max}^{(-)}}{L_{max}^{GF(-)}}$	$\frac{W^F}{W^{CF}}$
3	2+1	0.9 s	2.145	0.500	-5.1%	-4.2%	-10.1%
4	2+2	0.8 s	2.477	0.333	+3.8%	-22.6%	-10.8%
5	3+2	7.3 s	2.574	0.250	+4.6%	-26.4%	-14.0%
6	3+3	7.3 s	2.721	0.200	-5.1%	-25.5%	-24.4%
7	3+2+2	10.6 s	2.898	0.167	-10.5%	-16.8%	-27.5%
8	3+3+2	10.0 s	2.969	0.143	+8.3%	-39.0%	-19.1%
9	3+3+3	17.0 s	3.179	0.125	-1.6%	-7.4%	-22.2%
10	3+3+2+2	16.1 s	3.330	0.111	-0.6%	-11.5%	-26.1%
11	3+3+3+2	35.2 s	3.504	0.100	+1.2%	-18.8%	-24.2%
12	3+3+3+3	21.5 s	3.516	0.091	-1.4%	-12.0%	-25.0%
13	4+3+3+3	87.1 s	3.641	0.083	-7.0%	-12.6%	-27.3%
14	4+4+3+3	47.1 s	3.810	0.077	-8.6%	-17.8%	-29.5%
15	4+4+4+3	95.6 s	3.987	0.071	-10.4%	-17.1%	-29.7%
16	4+4+4+4	78.0 s	4.068	0.067	-14.7%	-13.0%	-35.5%

(a) Three-step approach.

K	chunking	time$_{W^F}$	$\frac{W^F}{V}$	$L_{max}^{(-)}$	$\frac{W^F}{W^{GF}}$	$\frac{L_{max}^{(-)}}{L_{max}^{GF(-)}}$	$\frac{W^F}{W^{CF}}$
4	2+2	3.3 s	2.293	0.333	-3.9%	-22.6%	-17.4%
5	3+2	10.3 s	2.447	0.250	-0.6%	-26.4%	-18.2%
8	3+3+2	L 8.7 s	L 2.700	L 0.143	L -1.5%	L -0.0%	L -26.4%
11	3+3+3+2	LT 60.5 s	LT 3.131	LT 0.100	LT -9.6%	LT -0.0%	LT -32.3%

(b) Two-step approach.

workload limits. The three-step approach performs slightly worse, still providing solutions with an 8 - 22% lower memory consumption than the chaining approach. Compared to the greedy failure approach, the three-step approach has a higher replication factor for $K > 3$, but its worst-case workload share is more than 20% lower. To control the worst-case workload share, we prefer the two-step approach because we can directly set the limit in the final step. In contrast, for the three-step approach, we may have to adapt the limit in step 2, in which node-failure robustness is already ensured on the chunk level.

For TPC-DS, the calculation time of our two-step approach becomes relatively high for larger cluster sizes K (i.e., multiple minutes for $K > 9$) due to the (more) complex optimal data enhancement in step 2 (compared to step 3 of the three-step approach). Hence, we focus on the three-step approach, for which Table 6.12a shows the results. Table 6.12b shows selected results (i.e., for $K = 4, 5$ with $L_{max}^{(-)} = \frac{1}{K-1}$ and for $K = 8, 11$ with $L_{max}^{(-)} := L_{max}^{GF(-)}$) of our two-step algorithm with a lower memory consumption than the greedy failure approach.

Naturally, the calculation times are higher than for TPC-H. Still, we were able to calculate all three-step solutions in 96 s or less without query clustering and solver relaxations. For all shown TPC-DS results, we only used a time limit (of 30 s for step 2) for the two-step solution with K=11 nodes and a relaxed failure load.

We observe that our three-step solutions lower the worst-case workload limits compared to the greedy failure approach by 4 - 39% while having a larger replication factor only for $K = 4, 5, 8, 11$. For $K = 4, 5$, we were able to outperform the greedy approach with regard to the memory consumption by using our two-step approach (see Table 6.12b), still using the optimal worst-case workload limit. Using the worst-case limit of the greedy approach, the solutions of our two-step approaches have a lower memory consumption than the greedy failure approach (see Table 6.12b, $K = 8, 11$). These results underline our model's flexibility: By simply changing the worst-case workload limit, we are able to tune our solutions, being more memory-efficient while sacrificing load-balancing in failure cases.

Compared to the chaining approach (with the same worst-case workload limits), our three-step approach requires 10 - 36% less memory for $3 \leq K \leq 16$. We achieved the highest load and memory improvements for $K = 8$ and $K = 16$, respectively, for which optimal solutions were intractable. Besides the comparison to optimal solutions, we find that these results indicate the quality of our ILP-based heuristic.

For the accounting workload, we found that the pure three-step approach requires too high runtimes. Hence, we combined it with the query clustering approach, *fixedly* assigning $|Q^F| = 4361$ queries that account for a small workload share to a single node. There are $|Q^R| = 100$ *remaining* queries for distributing and ensuring load-balancing flexibility for all single-node failure cases. Table 6.13 shows the results of our ILP-based heuristic, combining the query clustering and three-step approach.

Using our heuristic, the maximum calculation time was 63 s for $K = 10$ (while not using a time limit yet). All other solutions were obtained in 20 s or less. For the accounting workload, the calculation times were dominated by the optimal fragment enhancement, i.e., step 3.

With regard to the memory consumption, our heuristic solutions outperform the greedy failure and chaining approach. In contrast to TPC-H and TPC-DS, the solutions of the greedy approach achieve a perfect load balancing in failure cases for all $K \leq 16$ except $K = 5$. As a result, the memory savings of our ILP-based heuristic over the greedy failure approach are significantly larger ($-37\% < \frac{W^F}{W^{GF}} < -12\%$). Compared to the chaining approach, we require even 22 - 41% less memory.

Table 6.13.: ILP-based heuristic solutions (three-step and query clustering) that consider **node failures** for the **accounting workload** with different cluster sizes K: number of allocated $|Q^R|$ and fixed $|Q^F|$ queries, chunk decomposition, calculation time, replication factor $\frac{W^F}{V}$, (optimal) worst-cast workload share for all K single-node failure cases, relative memory consumption $\frac{W^F}{W^{GF}}$ and relative worst-case workload share improvement $\frac{L^{(-)}_{max}}{L^{GF(-)}_{max}}$ compared to the greedy failure approach [181], and relative memory consumption $\frac{W^F}{W^{CF}}$ compared to the chaining approach [108].

K	$\lvert Q^R\rvert + \lvert Q^F\rvert$	chunking	time$_{W^F}$	$\frac{W^F}{V}$	$L^{(-)}_{max}$	$\frac{W^F}{W^{GF}}$	$\frac{L^{(-)}_{max}}{L^{GF(-)}_{max}}$	$\frac{W^F}{W^{CF}}$
3	100+4361	2+1	0.4 s	2.336	0.500	-12.1%	-0.0%	-22.1%
4	100+4361	2+2	0.5 s	3.010	0.333	-17.7%	-0.0%	-24.8%
5	100+4361	3+2	1.1 s	3.284	0.250	-15.3%	-1.9%	-29.5%
6	100+4361	3+3	1.7 s	3.653	0.200	-19.5%	-0.0%	-31.3%
7	100+4361	4+3	3.3 s	4.155	0.167	-24.9%	-0.0%	-34.2%
8	100+4361	4+4	2.3 s	4.675	0.143	-28.5%	-0.0%	-36.1%
9	100+4361	3+3+3	2.8 s	5.214	0.125	-23.0%	-0.0%	-30.9%
10	100+4361	4+3+3	63.3 s	5.598	0.111	-27.8%	-0.0%	-34.4%
11	100+4361	4+4+3	8.1 s	5.939	0.100	-29.4%	-0.0%	-35.4%
12	100+4361	3+3+3+3	19.8 s	6.092	0.091	-35.3%	-0.0%	-40.2%
13	100+4361	4+3+3+3	12.9 s	6.624	0.083	-36.4%	-0.0%	-40.8%
14	100+4361	4+4+3+3	8.4 s	7.179	0.077	-33.3%	-0.0%	-38.3%
15	100+4361	4+4+4+3	4.9 s	7.415	0.071	-32.3%	-0.0%	-37.3%
16	100+4361	4+4+4+4	18.0 s	7.655	0.067	-29.7%	-0.0%	-35.7%

Summary: *Our numerical evaluation shows that our extension for node failures outperforms both existing approaches when comparing the combination of (i) memory consumption (up to 41% better) and (ii) worst-case workload limit (up to 45% better). Even large problems can be solved in a reasonable amount of time (< 100 s). Finally, our memory-efficient approaches can always guarantee an optimal load balancing. The quality of our results is based on the mutually supportive and flexible interplay of three ILP models, (i) the memory-efficient workload decomposition, (ii) the optimal robust model for node failures, and (iii) optimal data enhancements.*

End-to-End Evaluation and Summary

In this section, we evaluate allocations that consider node failures for TPC-H and TPC-DS in end-to-end experiments. Further, we summarize our results.

For each number of nodes K and each allocation approach, we evaluate $K + 1$ scenarios: one scenario without a node failure and K scenarios in which node k failed, i.e., node k is not used for query processing. Figure 6.6e and 6.6f show the TPC-H and TPC-DS query throughput without a failure (regular) and the measured minimum (worst-case) performance of all failure scenarios. Figure 6.6c and 6.6d

show the corresponding normalized numerical throughputs, which we derived from the worst-case workload limit: For the greedy failure approach, it is $\frac{L_{max}^{(-)}}{L_{max}^{GF(-)}} \cdot (K-1)$; for all other approaches, it is $K-1$. Normalized numerical throughputs for the accounting workload are shown in Table 6.7b. Figure 6.6a, 6.6b, and 6.7a visualize the memory consumption of allocations that consider node failures for the three evaluated workloads.

The end-to-end throughput corresponds to the numerical results: For the chaining, two-step, three-step, and optimal failure approach, we observe that no matter which node fails, the throughput of these allocations remains (more or less) stable. The stable performance indicates that for each single-node failure, the workload can be distributed in a way that no cluster node is overloaded disproportionally. Particularly, we avoid worst-case scenarios: Having a failed node whose workload share can only be taken over by a single node that itself cannot offload its assigned queries, the node's workload share would double, resulting in a bottleneck for query streams *halving* the throughput.

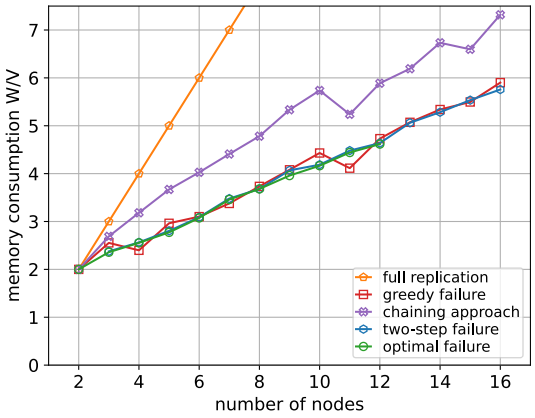
The worst-case throughput of the chaining, two-step, three-step, and optimal failure approach is considerably better than the throughput of the greedy failure approach, which may decrease significantly for specific cluster sizes K and failure scenarios (see TPC-H for $K = 11, 15$ and TPC-DS for $K = 8, 14$). Naturally, the end-to-end results do not perfectly match the numerical results because query execution times may differ from modeled query costs (for example through caching effects), which is noticeable in long-running experiments. Nevertheless, the theoretical results mostly indicate which greedy solutions have a better (e.g., TPC-H for $K = 9, 16$ and TPC-DS for $K = 9$) or worse (e.g., TPC-H for $K = 11, 14, 15$ and TPC-DS for $K = 8, 14$) worst-case limit.

Finally, our measurements show a higher throughput of allocations that consider node failures compared to basic solutions (see Figure 6.5b and 6.5d) in cases without failures. By ensuring that all queries are executable by multiple nodes, allocations become more robust with regard to inaccurate model costs and query timings because we have greater flexibility to distribute queries among cluster nodes.

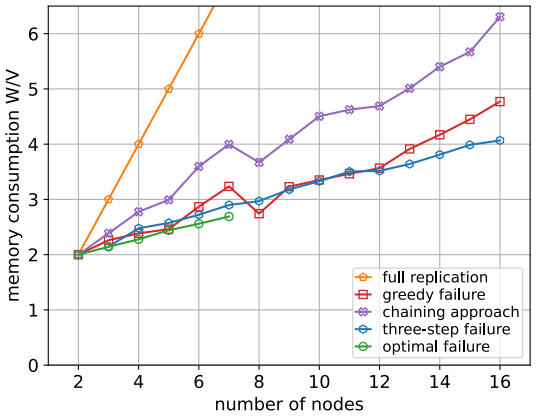
The visualized memory consumptions show that our heuristic's memory consumption is near-optimal. Most importantly, our ILP-based heuristic remains tractable for larger cluster sizes K and more complex workloads.

6.3.2. Workload Uncertainty

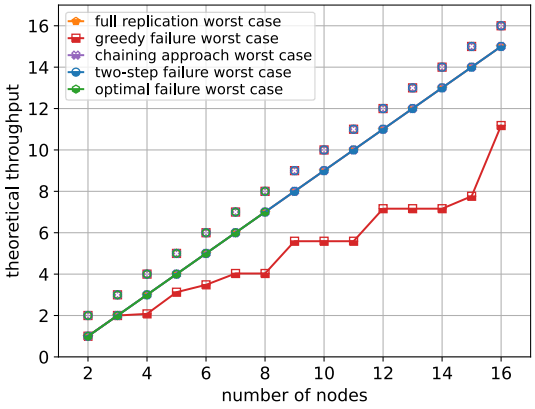
In the following, we compare fragment allocation approaches that consider multiple input scenarios, e.g., for coping with workload uncertainty. We evaluate ILP-based approaches (see Section 5.3.2) and the greedy "merge" approach (see Section 4.3.1) numerically for TPC-H, TPC-DS, and the accounting workload. Both ILP-based approaches and the greedy merge approach allow a flexible number of input scenarios $S \geq 1$. Each workload scenario is characterized by a query distribution, having different query subsets and query frequencies. For workload scenario $s = 1$, we set



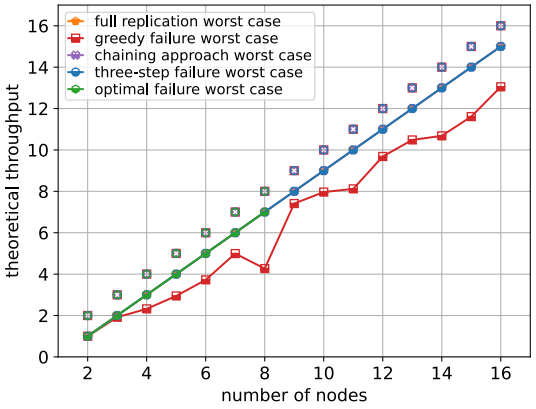
(a) TPC-H: memory consumption.



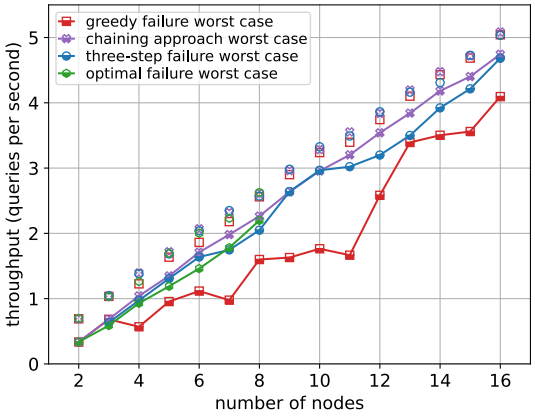
(b) TPC-DS: memory consumption.



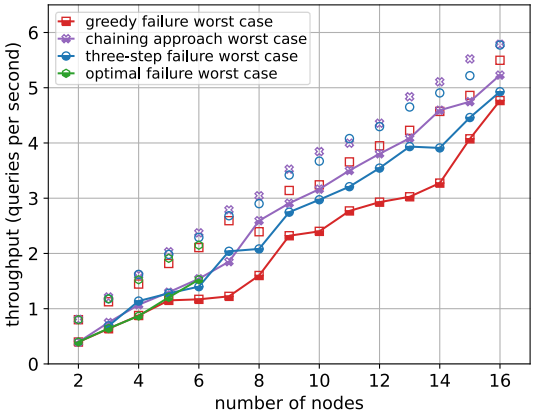
(c) TPC-H: numerical throughput.



(d) TPC-DS: numerical throughput.

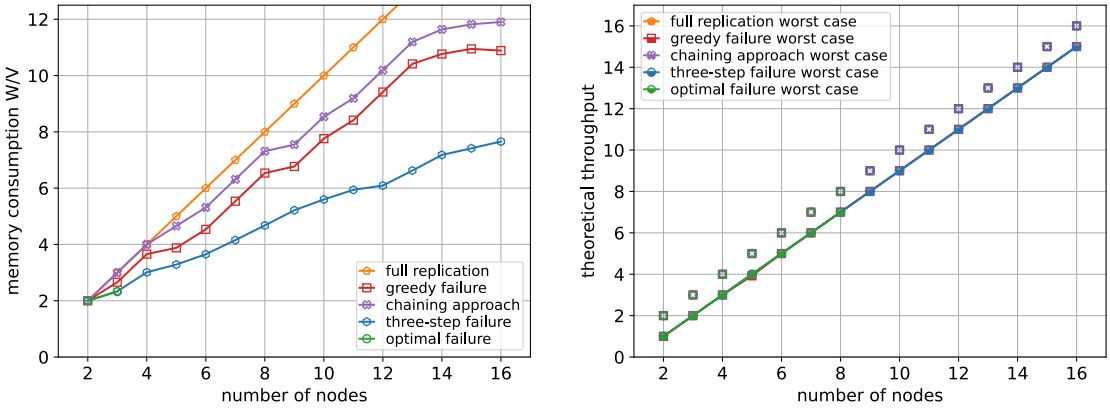


(e) TPC-H: end-to-end throughput.



(f) TPC-DS: end-to-end throughput.

Figure 6.6.: Evaluation of allocations that consider **node failures** for TPC-H and TPC-DS database cluster with 2 - 16 nodes: memory consumption, numerical throughput, and end-to-end throughput for the greedy failure approach, chaining approach, heuristic ILP-based approaches, and optimal failure solutions.



(a) Accounting: memory consumption. **(b)** Accounting: numerical throughput.

Figure 6.7.: Evaluation of allocations that consider **node failures** for the accounting workload with 2 - 16 nodes: memory consumption and numerical throughput for full replication, the greedy failure approach, chaining approach, heuristic ILP-based approaches, and optimal failure solutions.

the query frequency $f_{j,1} := 1$ for all queries $j = 1, ..., Q$. For all other scenarios, we use randomly generated 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 and, on average, we have $E(f_{j,s}) = 1$, $j = 1, ..., Q$, $s = 2, ..., S$. In our evaluation, we use $p = 0.75$.

Recap, given the S input scenarios, our ILP-based approaches for workload uncertainty and the greedy merge approach calculate allocations with an even load balancing for each scenario $s = 1, ..., S$. We also evaluate allocations against randomized "unseen" workload scenarios $\tilde{s} = 1, ..., \tilde{S}$, which we generated in the same way as input scenarios. We use our basic ILP model (5.1) - (5.5) with the penalty approach (see Section 5.1.1) and a fixed fragment allocation for calculating the maximum load $\tilde{L}$ that a single node must take for each given unseen scenario $\tilde{s} = 1, ..., \tilde{S}$ (similar to the evaluation of the greedy failure approach). The *average* worst-case workload share over all $\tilde{S}$ unseen scenarios is denoted by $E(\tilde{L})$, which is bounded from below by the best possible value $\frac{1}{K}$ (when the load can be perfectly balanced for each scenario). Further, we denote the expected numerical throughput by $E(\frac{1/K}{\tilde{L}})$, i.e., the average over all ratios of the optimal and the worst-case workload share.

All scenario-specific query distributions are available online [99]. In the following, we first present a detailed (introductory) evaluation for TPC-DS with $K = 8$ nodes. Then, we present results of all three workloads for all cluster sizes $2 \leq K \leq 16$.

Detailed Numerical Evaluation for TPC-DS with $K = 8$ Nodes

In this section, we evaluate different ILP-based approaches and the greedy merge approach for an increasing number of input scenarios $S \geq 1$ using TPC-DS with $K = 8$ nodes as a detailed representative example. We investigate the solution quality with regard to two properties: (i) We evaluate the required memory consumption W^U

for enabling an even load balancing for a fixed number of input scenarios S. (ii) We evaluate the load balancing of allocations against out-of-sample workloads. Therefore, we state the difference of the average worst-case workload share over all unseen scenarios $E(\tilde{L})$ and the optimal value $\frac{1}{K}$. Further, we derive the expected numerical throughput $E(\frac{1/K}{\tilde{L}})$.

For different numbers of input scenarios $S \leq 100$, Table 6.14a shows the results of ILP-based approaches, i.e., optimal ($|Q^F| = 0$, (no) chunking: 8) and heuristic ($|Q^F| = 0, 47$ clustered queries, (decomposition) chunking: $4 + 4$) solutions. Table 6.14b summarizes the solution properties of allocations from the greedy merge approach. Both tables list the number of input scenarios S, the allocation's calculation time, normalized memory consumption $\frac{W^U}{V}$, the difference $E(\tilde{L}) - \frac{1}{K}$, and the expected numerical throughput $E(\frac{1/K}{\tilde{L}})$.

Using the number of fixed queries $|Q^F|$ and the decomposition's chunking, we can (indirectly) control the calculation time. Optimal solutions do not scale but naturally provide the lowest memory consumption $\frac{W^U}{V}$. The partial clustering and decomposition approach are effective and allow deriving allocations for dozens of scenarios S quickly. The required amount of data $\frac{W^U}{V}$ increases with S. The increase is monotonous for the optimal solution (and merge approach). However, it does not have to be monotonous when using a decomposition because intermediate allocations (which provide the input for the following splits) may differ.

Our replication factors $\frac{W^U}{V}$ are significantly below K (i.e., better than full replication). Further, for the same number of input scenarios S, we achieve lower replication factors than the greedy merge approach (e.g., $\frac{W^U}{W^{GU}} < -35\%$ for $S = 5$). The improvements are greater for a larger number of input scenarios S, which demonstrates that merging entire nodes strongly increases the memory consumption and, thus, loses optimization potential when optimizing for specific input scenarios. However, the increasing memory consumption is not (necessarily) wasted when optimizing against uncertain workloads. Hence, we evaluate whether the increasing memory consumption effectively helps to improve the load balancing for unseen workloads.

If more scenarios S are taken into account, the worst-case workload shares $\tilde{L}$ (over unseen scenarios) decrease/improve. For TPC-DS with $K = 8$, considering $S = 20$ (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 $E(\tilde{L}) - \frac{1}{K} \leq 0.0076$, which is by far better than the basic $S = 1$ solution (with $E(\tilde{L}) - \frac{1}{K} = 0.0904$ for $|Q^F| = 0$) that optimizes only for a single scenario. Naturally, the higher robustness with $S = 20$ is achieved by using more data, i.e., a higher replication factor of 2.886 (in 6 s, $|Q^F| = 47$) instead of 1.902 ($|Q^F| = 0$) when using only one scenario ($S = 1$). Compared to achieving robustness via full replication (with $\frac{W}{V} = K = 8$), this is remarkable.

Further, we can evaluate the throughput robustness against unseen workloads for allocations of the greedy merge approach (see Table 6.14b) depending on the amount of allocated data $\frac{W}{V}$. For $S = 2$, we obtained the optimality gap $E(\tilde{L})^{GU} -$

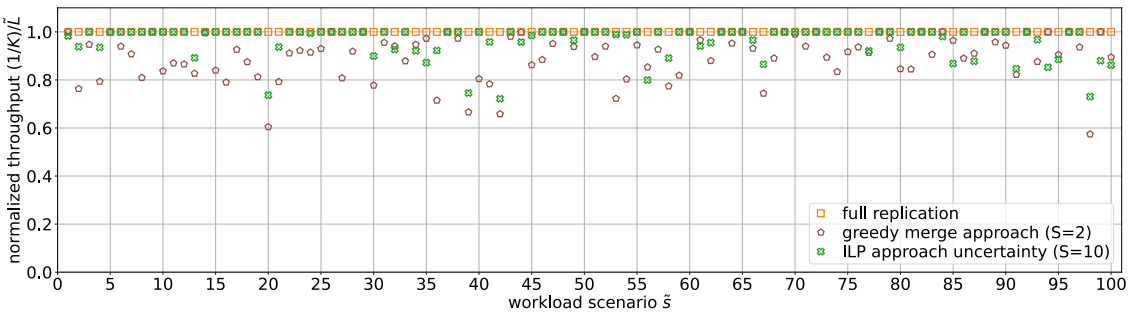
Table 6.14.: Approaches that consider **workload uncertainty** using multiple input scenarios S: number of allocated $|Q^R|$ and fixed $|Q^F|$ queries, and chunk decomposition for ILP-based approaches; calculation time and memory consumption $\frac{W^U}{V}$; difference of expected $E(\tilde{L})$ and optimal load limit $\frac{1}{K}$, and expected numerical throughput $E(\frac{1/K}{\tilde{L}})$ for $\tilde{S} = 100$ unseen workload scenarios.

$\lvert Q^R\rvert + \lvert Q^F\rvert$	chunking	S	time$_{W^U}$	$\frac{W^U}{V}$	$\frac{W^U}{W^{GU}}$	$E(\tilde{L}) - \frac{1}{K}$	$E\left(\frac{1/K}{\tilde{L}}\right)$
95+0	8	1	484.9 s	1.852	-19.7%	0.0899	0.605
95+0	8	2	2097.9 s	1.948	-30.3%	0.0782	0.649
95+0	8	3	2611.4 s	2.098	-30.8%	0.0350	0.805
95+0	8	4	5966.3 s	2.189	-36.8%	0.0326	0.818
95+0	8	5	17201.9 s	2.285	-38.4%	0.0234	0.860
95+0	4+4	1	24.1 s	1.902	-17.5%	0.0904	0.605
95+0	4+4	2	13.7 s	2.030	-27.4%	0.0717	0.670
95+0	4+4	3	12.6 s	2.241	-26.1%	0.0452	0.760
95+0	4+4	4	10.8 s	2.320	-32.9%	0.0286	0.837
95+0	4+4	5	20.9 s	2.391	-35.6%	0.0235	0.862
95+0	4+4	7	32.5 s	2.429	-40.0%	0.0148	0.909
95+0	4+4	10	77.0 s	2.483	-41.2%	0.0123	0.925
95+0	4+4	20	567.5 s	2.700	-44.7%	0.0076	0.951
95+0	4+4	30	277.7 s	2.658	-50.0%	0.0051	0.968
95+0	4+4	40	737.5 s	2.752	-50.7%	0.0041	0.975
95+0	4+4	50	663.4 s	2.801	-50.1%	0.0040	0.975
48+47	4+4	1	0.8 s	2.059	-10.7%	0.0823	0.628
48+47	4+4	2	1.0 s	2.124	-24.0%	0.0736	0.662
48+47	4+4	3	0.9 s	2.287	-24.5%	0.0471	0.751
48+47	4+4	4	1.4 s	2.418	-30.1%	0.0279	0.836
48+47	4+4	5	1.3 s	2.403	-35.3%	0.0270	0.841
48+47	4+4	7	2.4 s	2.643	-34.7%	0.0140	0.914
48+47	4+4	10	3.0 s	2.696	-36.2%	0.0155	0.904
48+47	4+4	20	6.0 s	2.886	-40.9%	0.0055	0.963
48+47	4+4	30	11.7 s	2.964	-44.3%	0.0039	0.976
48+47	4+4	40	13.6 s	2.896	-48.2%	0.0031	0.981
48+47	4+4	50	27.7 s	2.913	-48.2%	0.0023	0.985
48+47	4+4	100	45.3 s	2.989	-47.7%	0.0019	0.988

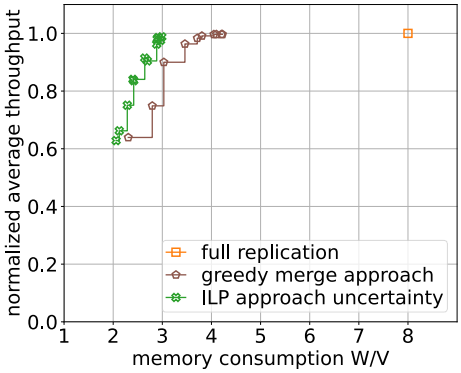
(a) ILP-based approaches.

S	time$_{W^{GU}}$	$\frac{W^{GU}}{V}$	$E(\tilde{L})^{GU} - \frac{1}{K}$	$E\left(\frac{1/K}{\tilde{L}}\right)^{GU}$
1	< 1 s	2.307	0.0789	0.639
2	< 1 s	2.795	0.0501	0.749
3	< 1 s	3.031	0.0157	0.900
4	< 1 s	3.460	0.0053	0.963
5	< 1 s	3.712	0.0023	0.983

(b) Greedy merge approach.



(a) Normalized throughput $\frac{1/K}{\tilde{L}}$ for all $\tilde{S} = 100$ individual unseen scenarios $\tilde{s} = 1, ..., \tilde{S}$.



(b) Memory consumption $\frac{W}{V}$ vs. normalized average throughput $E(\frac{1/K}{\tilde{L}})$ over $\tilde{S} = 100$ unseen scenarios.

Figure 6.8.: Evaluation of allocations considering **multiple workloads** for TPC-DS with $K = 8$ nodes: memory consumption and throughput of $\tilde{S} = 100$ unseen workload scenarios for allocations using different numbers of input scenarios S.

$\frac{1}{K} = 0.0501$ and replication factor $\frac{W^{GU}}{V} = 2.795$. Compared to that, our $S = 10$ solution with $|Q^F| = 47$ fixed queries provides a *clearly better robustness* (i.e., $E(\tilde{L}) - \frac{1}{K} = 0.0155 < 0.0501 = E(\tilde{L})^{GU} - \frac{1}{K}$) and requires *less memory* (i.e., $\frac{W^U}{V} = 2.696 < 2.795 = \frac{W^{GU}}{V}$). Figure 6.8a shows the corresponding individual numerical throughput $\frac{1/K}{\tilde{L}}$ for all $\tilde{S} = 100$ unseen scenarios of these solutions and full replication.

The better combinations of memory consumption $\frac{W^U}{V}$ and robustness can be observed over the full range of input scenarios S. Figure 6.8b visualizes this result: For selected S, the figure shows the expected/average numerical throughput $E(\frac{1/K}{\tilde{L}})$ per memory consumption for the greedy merge approach, our ILP-based solution with $|Q^F| = 47$, and full replication.

Naturally, the specific results depend on the specific input parameters, including the randomized frequencies. However, using the following extensive evaluation with different workloads and cluster sizes, we show that the overall properties remain.

Numerical Evaluation for all Workloads and Summary

In the following, we evaluate allocation approaches for workload uncertainty for TPC-H, TPC-DS, and the accounting workload. Figure 6.9, 6.10, and 6.11 show the numerical throughput per memory consumption for all three workloads and cluster sizes $2 \leq K \leq 16$. The number of fixed queries $|Q^F|$ and decomposition can be chosen such that our approach remains applicable for larger problem sizes. For TPC-DS and the accounting workload, we show results of our decomposition approach with $|Q^F| = 47$ and $|Q^F| = 4361$ clustered queries. Using these numbers of fixed queries $|Q^F|$, we could calculate all allocations with $S \leq 50$ in under 60 s.

Compared to the merge approach, our approach outperforms its combinations of memory consumption and throughput robustness against uncertain workloads for all workloads. For TPC-H, our results are particularly better for small cluster sizes $K < 10$. For TPC-H and larger cluster sizes, the merge approach works well. The reason is that for these inputs ($Q = 22$, $K \geq 10$), the individual scenarios' allocations assign only a low number of queries to nodes. Merging such nodes naturally loses less optimization potential compared to cases with many queries per node.

For TPC-DS, we were able to find allocations with a clearly lower memory consumption and/or better numerical throughput than the merge approach.

For the accounting workload, we find that the optimality gaps $E(\tilde{L}) - \frac{1}{K}$ are, on average, lower while the replication factors $\frac{W}{V}$ are higher for all S. In these cases, a *smaller* number of input scenarios is necessary to obtain a certain throughput robustness against unseen workloads. In particular, the merge approach already reaches a close to optimal numerical throughput with $S = 3$ input scenarios.

For all workloads, we achieved a good trade-off between throughput robustness and memory consumption by using 3 - 100 randomized input scenarios S (depending on the targets). Further, we find our approach better for fine-tuning towards specific targets (e.g., a specific maximum memory consumption) because merging entire nodes may abruptly change an allocation's property. Note, using ILP, we could further tune the trade-off between memory consumption and load limits by using a penalty approach, not strictly enforcing an optimal load limit for each input scenario. Moreover, we can flexibly adjust the number of fixed queries, decomposition, and time limits to improve the solution quality using a longer calculation time.

Summary: *We find that optimizing an allocation for only a single expected workload is not robust against unseen workloads. We can extend our basic model to calculate memory-efficient allocations that enable an even load balancing for multiple workload scenarios. The memory efficiency of our ILP-based approach allows for including more scenarios within a certain memory budget than the greedy merge approach. Further, our solutions provide better throughput robustness against unseen workloads using the same or even less data.*

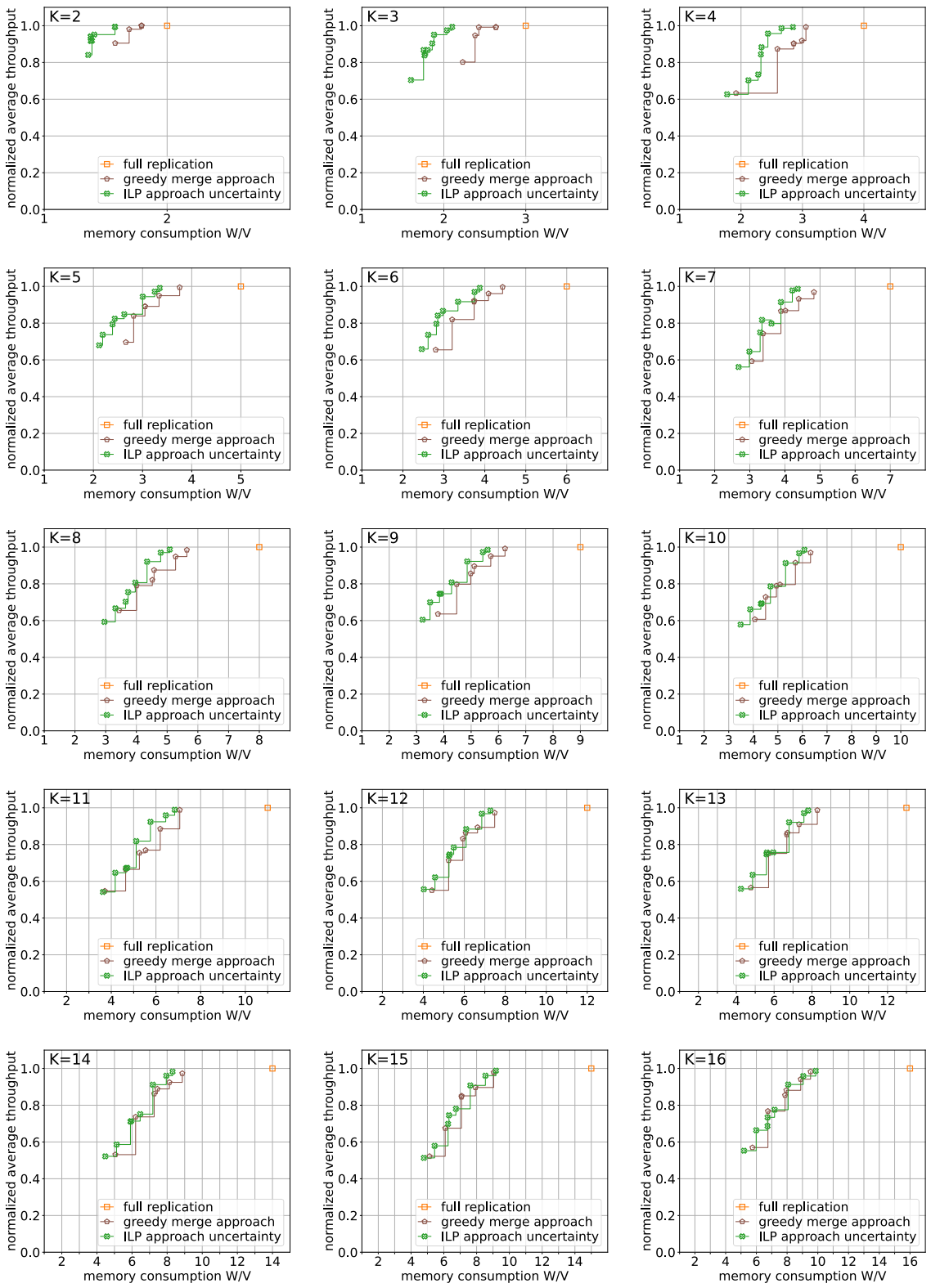


Figure 6.9.: Evaluation of allocations considering multiple workloads; TPC-H.

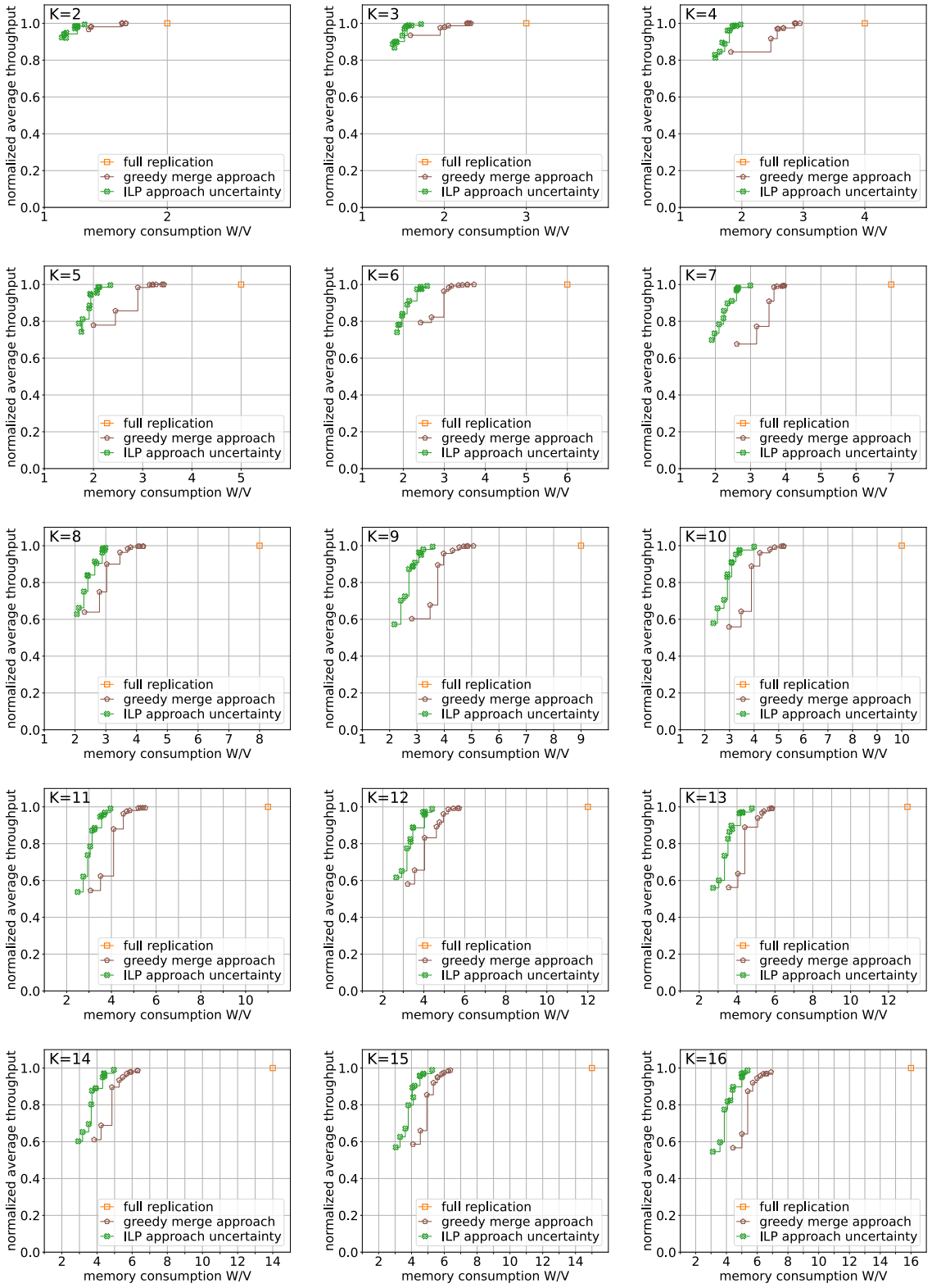


Figure 6.10.: Evaluation of allocations considering multiple workloads; TPC-DS.

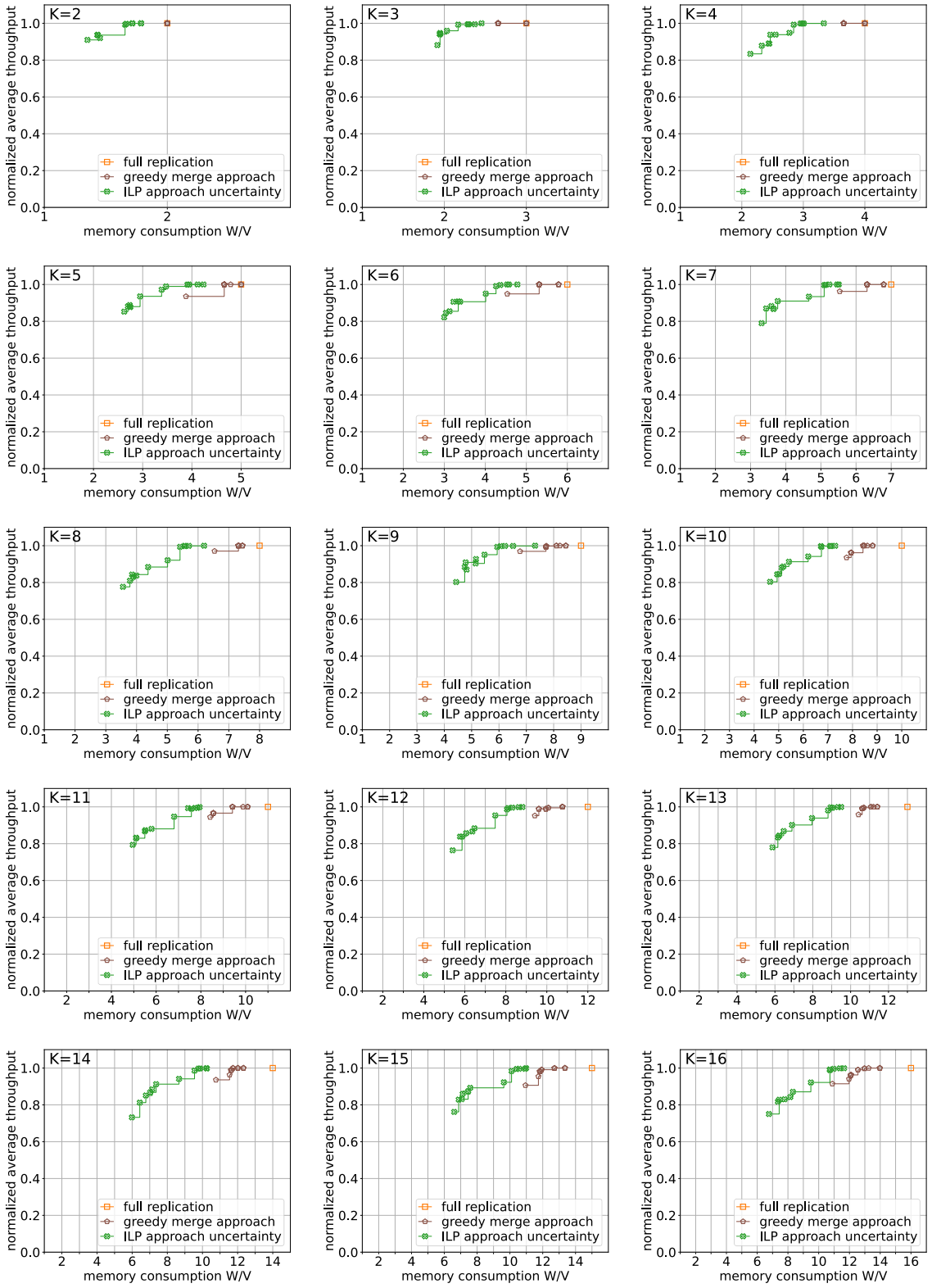


Figure 6.11.: Evaluation of allocations considering multiple workloads; accounting.

6.3.3. Data Modifications

In this section, we evaluate our allocation models that consider data modification costs (see Section 5.3.3) for TPC-H and TPC-DS. (Recap, the accounting trace is read-only.) We consider workloads with all TPC-H and the running 95 TPC-DS (read) queries, which account for 90% of the overall query load. The remaining 10% of the load comes from data modification queries, whose load is aligned to the benchmark's specification. We assume update queries that affect entire tables and consider logical data modification costs, i.e., as soon as a node stores a single data fragment of an updated table, the modification costs for the table occur at the node, and the node must store all table fragments. To better understand the following results, Table 6.15 summarizes the sizes for updated tables, the corresponding modification costs, and the share of the read-only workload accessing the updated tables.

We want to balance the workload evenly among the cluster nodes while considering data modification costs. Table 6.16 shows the solutions of our optimal model and compares them to the greedy approach. Table 6.17 shows solutions that we calculated using ILP-based heuristic for TPC-DS in about 10 s or less, clustering 47 workload-unintense queries to a single node and using the solver's time limit. Figure 6.12 visualizes the results of all approaches including full replication.

The memory consumption is considerably higher than for the read-only case (see Figure 6.5a and 6.5c) because the updated tables are large and accessed by a large read-only workload share (see Table 6.15) and, thus, must be replicated to many nodes. Overall, the greedy approach offers limited advantages over full replication: The memory consumption is at most 12% lower than for full replication. Only for TPC-DS and the cluster sizes $K = 12, 14, 16$, the data modification costs are reduced substantially (by around 30%). We find that the working principle of the greedy heuristic (see Section 4.3.1) to repeatedly increase the load limits to account for an even load balancing has optimization potential.

The ILP approach guarantees optimal solutions. For TPC-H, we could calculate optimal solutions in 2 s or less (which is faster than for the read-only case (see Table 6.2)) because the `lineitem` and `orders` tables must be replicated to many

Table 6.15.: Summary of workload tables with data modifications: table name, relative table size, relative load of data modifications, and the load share of read queries accessing the table.

table	size	modifications	read share
lineitem	60%	9%	89%
orders	28%	1%	68%

(a) TPC-H.

table	size	modifications	read share
catalog_sales	29%	1.1%	57%
catalog_returns	3%	0.1%	21%
inventory	10%	5.6%	12%
store_sales	36%	2.3%	82%
store_returns	4%	0.2%	12%
web_sales	15%	0.6%	51%
web_returns	2%	0.1%	11%

(b) TPC-DS.

Table 6.16.: Optimal solutions of the model that considers **modification costs** for different workloads and cluster sizes K: calculation time, replication factor $\frac{W^M}{V}$, share of modification costs, and relative memory and modification cost savings compared to the greedy approach [181] $\frac{W^M}{W^{GM}}$.

K	time$_{W^M}$	$\frac{W^M}{V}$	modifications	$\frac{W^M}{W^{GM}}$	modifications
2	0.0 s	1.894	10.0%	-2.0%	0.0%
3	0.1 s	2.782	10.0%	-0.3%	0.0%
4	0.2 s	3.433	9.8%	-0.6%	0.0%
5	0.2 s	4.321	9.8%	-7.1%	-2.0%
6	0.4 s	5.171	9.8%	-1.8%	0.0%
7	0.6 s	5.821	9.7%	-11.9%	-2.9%
8	0.6 s	6.719	9.8%	-10.1%	-2.5%
9	0.7 s	7.026	8.8%	-16.7%	-12.2%
10	0.8 s	7.919	8.9%	-14.6%	-11.0%
11	0.9 s	8.525	8.9%	-17.6%	-10.9%
12	1.2 s	9.451	9.0%	-15.0%	-10.0%
13	1.2 s	10.344	9.1%	-13.8%	-9.2%
14	1.4 s	10.949	9.1%	-17.5%	-9.3%
15	1.7 s	11.985	9.1%	-14.6%	-8.7%
16	1.9 s	12.728	9.2%	-13.4%	-7.5%

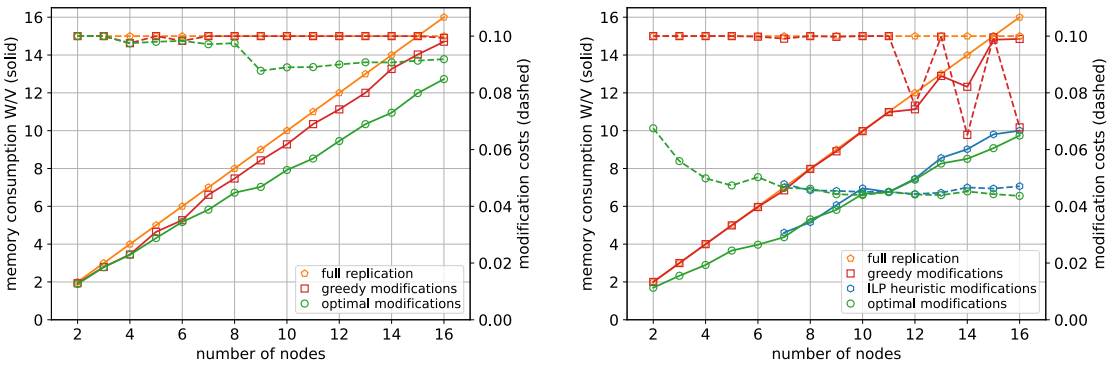
(a) TPC-H: N=61, Q=22, commercial database system, scale factor 10.

K	time$_{W^M}$	$\frac{W^M}{V}$	modifications	$\frac{W^M}{W^{GM}}$	modifications
2	0.1 s	1.693	6.8%	-15.3%	-32.5%
3	0.9 s	2.328	5.6%	-22.4%	-44.1%
4	3.7 s	2.899	5.0%	-27.5%	-50.1%
5	4.5 s	3.658	4.7%	-26.7%	-52.7%
6	8.0 s	3.969	5.0%	-33.4%	-49.6%
7	16.2 s	4.368	4.6%	-36.2%	-53.1%
8	16.5 s	5.310	4.6%	-33.4%	-53.8%
9	23.0 s	5.815	4.4%	-34.7%	-55.6%
10	29.4 s	6.651	4.4%	-33.3%	-56.0%
11	4872.3 s	6.750	4.5%	-38.5%	-54.9%
12	3601.9 s	7.407	4.4%	-33.4%	-41.5%
13	9840.2 s	8.262	4.4%	-35.9%	-56.0%
14	1341.6 s	8.512	4.5%	-30.9%	-30.6%
15	14455.0 s	9.074	4.4%	-38.7%	-55.5%
16	1634.4 s	9.743	4.4%	-34.4%	-35.6%

(b) TPC-DS: N=425, Q=95, commercial database system, scale factor 10.

Table 6.17.: ILP-based heuristic solutions that consider **modification costs** for TPC-DS with different cluster sizes K: calculation time, replication factor $\frac{W^M}{V}$, share of modification costs, and relative memory and modification cost savings compared to the greedy approach [181] $\frac{W^M}{W^{GM}}$. "C" indicates a solution using the clustering of 47 workload-unintense queries on one node, "T" indicates a solution using a time limit.

K	time$_{W^M}$	$\frac{W^M}{V}$	modifications	$\frac{W^M}{W^{GM}}$	modifications
7	CT 3.0 s	CT 4.609	CT 4.8%	CT-32.7%	CT-51.7%
8	CT 9.1 s	CT 5.159	CT 4.6%	CT-35.3%	CT-54.4%
9	CT 4.4 s	CT 6.065	CT 4.5%	CT-31.9%	CT-54.4%
10	CT 10.1 s	CT 6.953	CT 4.5%	CT-30.3%	CT-54.9%
11	CT 8.8 s	CT 6.758	CT 4.5%	CT-38.5%	CT-54.9%
12	CT 9.7 s	CT 7.458	CT 4.4%	CT-33.0%	CT-41.3%
13	CT 10.1 s	CT 8.556	CT 4.5%	CT-33.6%	CT-55.2%
14	CT 10.1 s	CT 9.018	CT 4.7%	CT-26.8%	CT-28.4%
15	CT 10.1 s	CT 9.803	CT 4.6%	CT-33.8%	CT-53.5%
16	CT 10.1 s	CT 10.000	CT 4.7%	CT-32.7%	CT-30.6%



(a) TPC-H. **(b)** TPC-DS.

Figure 6.12.: Evaluation of approaches that consider **modification costs** for a database cluster with 2 - 16 nodes: memory consumption and (relative) modification costs for full replication, the greedy approach, and ILP-based approaches.

nodes, which makes the allocation problem less complex. Compared to the greedy approach, we decrease the memory consumption by about 15% for larger cluster sizes while also reducing the modification costs by 10%.

TPC-DS offers greater optimization potential because the modification costs are spread across multiple tables, which are, on average, less frequently accessed by read queries (see Table 6.15b). The calculation time of optimal solutions is considerably higher than for TPC-H, reaching multiple hours for larger cluster sizes. However, we can quickly calculate near optimal solutions with our ILP-based heuristic (see Fig-

ure 6.12b). Compared to the greedy approach, we reduce the memory consumption by up to 39% while often *halving* the modification costs.

6.3.4. Reallocation Costs

Finally, we evaluate reallocation approaches numerically for (i) changing workloads and (ii) adding a new node to the cluster using TPC-H, TPC-DS, and the accounting workload. For both scenarios and all three workloads, we compare our ILP-based reallocation approaches against the greedy reallocation approach (i.e., the greedy basic approach in combination with the Hungarian algorithm (see Section 4.3.1)) and ILP-based basic approaches, which do not consider reallocation costs, in combination with the Hungarian algorithm.

Changing Workload

First, we evaluate reallocation approaches for changing workloads in a fixed-size cluster. To model a workload change, we defined two workload scenarios, which are characterized by different *overlapping* query subsets. For TPC-H, scenario 1 contains the queries 1 - 15, and scenario 2 contains the queries 8 - 22. For TPC-DS, scenario 1 includes queries of the range 1 - 66, and scenario 2 includes queries of the range 34 - 99. For the accounting workload (for which query IDs are ordered by query load), we partition the queries into three disjoint sets using the modulo operation; scenario 1 contains the query set $\{j \mid j \bmod 3 \neq 1, j = 1, ..., Q\}$ and scenario 2 contains the query set $\{j \mid j \bmod 3 \neq 2, j = 1, ..., Q\}$.

The initial allocation is optimized for workload scenario 1 and is calculated using our ILP-based approach. We want to calculate an allocation for scenario 2 without reallocating too much data. We compare solutions with regard to the memory consumption $\frac{W}{V}$ of the new allocation and the amount of reallocated data

$$R := \sum\nolimits_{i=1,...,N,k=1,...,K} a_i \cdot (1 - s_{i,k}) \cdot x_{i,k}, \tag{6.1}$$

which is the sum of added data for scenario 2; data deletion is assumed to have no/negligible costs. In the following tables and figures, we normalize the amount of reallocated data R by the amount of accessed data V.

Table 6.18 compares the solutions of our optimal reallocation model and optimal basic model to the greedy approach. Table 6.19 shows ILP-based heuristic solutions, which we calculated for more complex TPC-DS and accounting problem instances (with a high number of nodes K) in about 10 s or less, using our decomposition approach, the solver's time limit, and query clustering. Figure 6.13a, 6.13c, and 6.13e show the results of all approaches including full replication. Figure 6.14 visualizes exemplary allocations for TPC-DS and a cluster with $K = 4$ nodes.

Full replication is robust concerning workload changes and requires no reallocations. However, the memory consumption is significantly higher than those of partial replication approaches, particularly for larger cluster sizes.

Table 6.18.: Optimal solutions of data reallocation approaches for **changing workloads** for a database cluster with 2 - 16 nodes: calculation time, memory consumption $\frac{W^R}{V}$ and amount of reallocated data $\frac{R^R}{V}$ for the ILP-based *reallocation* approach, and relative memory $\frac{W^R}{W^{GR}}$ and reallocation costs $\frac{R^R}{R^{GR}}$ savings compared to the greedy approach; for the ILP-based *basic* approach, calculation time, and relative memory $\frac{W^*}{W^{GR}}$ and reallocation costs $\frac{R^*}{R^{GR}}$ savings compared to the greedy approach.

K	time$_{W^R}$	$\frac{W^R}{V}$	$\frac{R^R}{V}$	$\frac{W^R}{W^{GR}}$	$\frac{R^R}{R^{GR}}$	time$_{W^*}$	$\frac{W^*}{W^{GR}}$	$\frac{R^*}{R^{GR}}$
2	0.0 s	1.322	0.322	-13.0%	-17.1%	0.1 s	-13.0%	-17.1%
3	0.1 s	1.708	0.445	-11.5%	-41.0%	0.1 s	-16.3%	-28.2%
4	0.1 s	2.018	0.602	-10.0%	-27.6%	0.2 s	-11.7%	-21.4%
5	0.1 s	2.403	0.711	-2.4%	-30.5%	0.2 s	-9.3%	-20.6%
6	0.2 s	2.720	0.784	-5.7%	-19.9%	0.4 s	-9.8%	-17.5%
7	0.3 s	3.139	0.909	-0.2%	-21.6%	0.6 s	-13.4%	-11.3%
8	0.3 s	3.249	1.049	-3.2%	-13.6%	0.7 s	-8.3%	-8.1%
9	0.9 s	3.559	1.194	-3.5%	-16.3%	1.1 s	-7.3%	-15.3%
10	0.5 s	3.815	1.178	-4.6%	-20.5%	1.7 s	-9.7%	-12.7%
11	0.6 s	4.132	1.326	-11.6%	-25.1%	2.2 s	-11.6%	-24.3%
12	0.8 s	4.572	1.407	-10.6%	-31.0%	4.7 s	-13.2%	-29.4%
13	1.8 s	4.894	1.613	-6.9%	-19.6%	6.4 s	-10.8%	-11.2%
14	2.9 s	5.156	1.649	-7.1%	-26.2%	5.0 s	-12.3%	-21.3%
15	1.3 s	5.495	1.771	0.2%	-13.7%	6.6 s	-4.9%	-6.7%
16	2.6 s	5.647	1.882	-11.3%	-23.4%	8.6 s	-12.7%	-21.2%

(a) TPC-H: N=61, Q=22, commercial database system, scale factor 10.

K	time$_{W^R}$	$\frac{W^R}{V}$	$\frac{R^R}{V}$	$\frac{W^R}{W^{GR}}$	$\frac{R^R}{R^{GR}}$	time$_{W^*}$	$\frac{W^*}{W^{GR}}$	$\frac{R^*}{R^{GR}}$
2	0.1 s	1.029	0.148	-8.6%	-52.1%	0.2 s	-11.4%	-31.7%
3	0.1 s	1.161	0.208	-7.0%	-27.0%	0.8 s	-11.6%	5.0%
4	0.1 s	1.174	0.175	-29.6%	-77.2%	1.4 s	-29.7%	-73.9%
5	0.9 s	1.363	0.253	-25.1%	-67.0%	5.2 s	-29.2%	-66.8%
6	1.0 s	1.500	0.366	-18.6%	-53.3%	35.2 s	-21.8%	-50.2%
7	1.2 s	1.608	0.423	-15.9%	-46.8%	16.5 s	-20.1%	-43.3%
8	1.7 s	1.698	0.447	-11.1%	-38.4%	77.9 s	-14.8%	-34.0%
9	5.3 s	1.811	0.440	-22.6%	-57.3%	72.2 s	-26.4%	-54.5%
10	2.9 s	1.961	0.497	-25.8%	-57.9%	169.0 s	-29.9%	-49.8%
11	4.3 s	2.029	0.519	-23.8%	-57.0%	152.6 s	-26.8%	-48.4%
12	87.1 s	2.197	0.625	-31.8%	-60.4%	355.8 s	-34.9%	-57.4%
13	78.0 s	2.227	0.671	-18.5%	-43.9%	410.3 s	-21.2%	-41.3%
14	72.5 s	2.491	0.761	-18.9%	-44.3%	2288.5 s	-24.5%	-38.7%
15	72.5 s	2.510	0.788	-20.3%	-45.7%	781.6 s	-22.8%	-34.7%
16	191.4 s	2.626	0.791	-28.5%	-59.6%	2755.5 s	-32.7%	-52.7%

(b) TPC-DS: N=425, Q=95, commercial database system, scale factor 10.

K	time$_{W^R}$	$\frac{W^R}{V}$	$\frac{R^R}{V}$	$\frac{W^R}{W^{GR}}$	$\frac{R^R}{R^{GR}}$	time$_{W^*}$	$\frac{W^*}{W^{GR}}$	$\frac{R^*}{R^{GR}}$
2	9.0 s	1.549	0.315	-22.5%	-58.9%	7.8 s	-24.8%	-64.7%
3	24.8 s	1.980	0.440	-25.5%	-60.4%	103.6 s	-27.1%	-60.4%
4	141.7 s	2.359	0.524	-28.8%	-64.7%	7184.7 s	-30.0%	-63.2%
5	6336.3 s	2.916	0.922	-32.4%	-59.7%	-	-	-
6	630.3 s	3.265	0.891	-29.6%	-60.4%	-	-	-
7	1378.8 s	3.550	0.953	-40.5%	-71.6%	-	-	-

(c) Accounting workload: N=344, Q=4461, metadata.

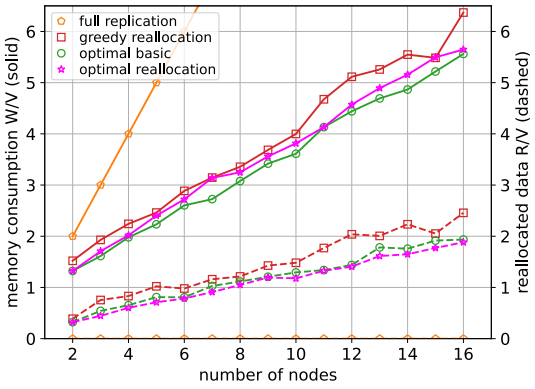
Table 6.19.: ILP-based heuristic solutions of data reallocation approaches for **changing workloads** for a database cluster with 2 - 16 nodes: calculation time, memory consumption $\frac{W^R}{V}$ and amount of reallocated data $\frac{R^R}{V}$ for the ILP-based *reallocation* approach, and relative memory $\frac{W^R}{W^{GR}}$ and reallocation costs $\frac{R^R}{R^{GR}}$ savings compared to the greedy approach; for the ILP-based *basic* approach, calculation time, and relative memory $\frac{W}{W^{GR}}$ and reallocation costs $\frac{R}{R^{GR}}$ savings compared to the greedy approach.. "$_{\text{D}}$" indicates a solution using the decomposition-based approach, "$_{\text{T}}$" indicates a solution using a time limit, "$_{\text{C}}$" indicates a solution using the clustering of workload-unintense queries on one node.

K	time$_{W^R}$	$\frac{W^R}{V}$	$\frac{R^R}{V}$	$\frac{W^R}{W^{GR}}$	$\frac{R^R}{R^{GR}}$	time$_W$	$\frac{W}{W^{GR}}$	$\frac{R}{R^{GR}}$
6	1.0 s	1.500	0.366	-18.6%	-53.3%	$_{\text{D}}$ 1.1 s	$_{\text{D}}$-20.5%	$_{\text{D}}$-46.8%
7	1.2 s	1.608	0.423	-15.9%	-46.8%	$_{\text{D}}$ 1.9 s	$_{\text{D}}$-19.4%	$_{\text{D}}$-43.5%
8	1.7 s	1.698	0.447	-11.1%	-38.4%	$_{\text{D}}$ 1.9 s	$_{\text{D}}$-12.4%	$_{\text{D}}$-32.3%
9	5.3 s	1.811	0.440	-22.6%	-57.3%	$_{\text{D}}$ 1.4 s	$_{\text{D}}$-23.9%	$_{\text{D}}$-49.0%
10	2.9 s	1.961	0.497	-25.8%	-57.9%	$_{\text{D}}$ 2.8 s	$_{\text{D}}$-28.3%	$_{\text{D}}$-44.9%
11	4.3 s	2.029	0.519	-23.8%	-57.0%	$_{\text{D}}$ 3.6 s	$_{\text{D}}$-24.1%	$_{\text{D}}$-50.5%
12	$_{\text{T}}$ 10.0 s	$_{\text{T}}$ 2.197	$_{\text{T}}$ 0.625	$_{\text{T}}$-31.8%	$_{\text{T}}$-60.4%	$_{\text{D}}$ 2.0 s	$_{\text{D}}$-32.8%	$_{\text{D}}$-56.4%
13	$_{\text{T}}$ 10.1 s	$_{\text{T}}$ 2.311	$_{\text{T}}$ 0.702	$_{\text{T}}$-15.4%	$_{\text{T}}$-41.2%	$_{\text{D}}$ 4.0 s	$_{\text{D}}$-19.0%	$_{\text{D}}$-39.1%
14	$_{\text{T}}$ 10.0 s	$_{\text{T}}$ 2.515	$_{\text{T}}$ 0.766	$_{\text{T}}$-18.1%	$_{\text{T}}$-43.9%	$_{\text{D}}$ 4.6 s	$_{\text{D}}$-22.8%	$_{\text{D}}$-37.8%
15	$_{\text{T}}$ 10.0 s	$_{\text{T}}$ 2.556	$_{\text{T}}$ 0.788	$_{\text{T}}$-18.9%	$_{\text{T}}$-45.7%	$_{\text{D}}$ 4.6 s	$_{\text{D}}$-20.9%	$_{\text{D}}$-38.8%
16	$_{\text{T}}$ 10.1 s	$_{\text{T}}$ 2.664	$_{\text{T}}$ 0.846	$_{\text{T}}$-27.4%	$_{\text{T}}$-56.8%	$_{\text{D}}$ 3.6 s	$_{\text{D}}$-29.9%	$_{\text{D}}$-50.2%

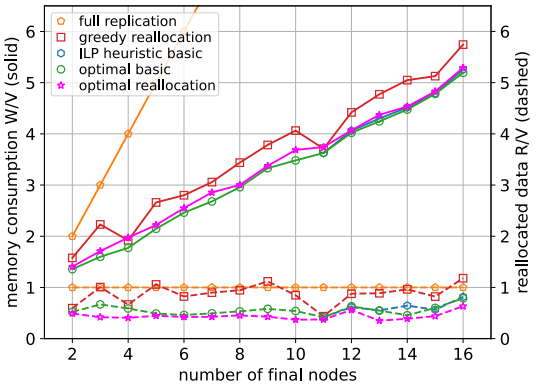
(a) TPC-DS: N=425, Q=95, commercial database system, scale factor 10.

K	time$_{W^R}$	$\frac{W^R}{V}$	$\frac{R^R}{V}$	$\frac{W^R}{W^{GR}}$	$\frac{R^R}{R^{GR}}$	time$_W$	$\frac{W}{W^{GR}}$	$\frac{R}{R^{GR}}$
2	$_{\text{C}}$ 0.1 s	$_{\text{C}}$ 1.552	$_{\text{C}}$ 0.317	$_{\text{C}}$-22.4%	$_{\text{C}}$-58.5%	$_{\text{C}}$ 0.1 s	$_{\text{C}}$-23.8%	$_{\text{C}}$-57.8%
3	$_{\text{C}}$ 0.2 s	$_{\text{C}}$ 1.938	$_{\text{C}}$ 0.442	$_{\text{C}}$-27.0%	$_{\text{C}}$-60.2%	$_{\text{C}}$ 0.2 s	$_{\text{C}}$-27.1%	$_{\text{C}}$-60.4%
4	$_{\text{C}}$ 0.7 s	$_{\text{C}}$ 2.394	$_{\text{C}}$ 0.560	$_{\text{C}}$-27.7%	$_{\text{C}}$-62.3%	$_{\text{C}}$ 0.8 s	$_{\text{C}}$-29.1%	$_{\text{C}}$-58.9%
5	$_{\text{C}}$ 1.2 s	$_{\text{C}}$ 2.951	$_{\text{C}}$ 0.932	$_{\text{C}}$-31.6%	$_{\text{C}}$-59.3%	$_{\text{C}}$ 5.3 s	$_{\text{C}}$-33.3%	$_{\text{C}}$-58.4%
6	$_{\text{C}}$ 1.7 s	$_{\text{C}}$ 3.216	$_{\text{C}}$ 0.914	$_{\text{C}}$-30.6%	$_{\text{C}}$-59.4%	$_{\text{CD}}$ 0.3 s	$_{\text{CD}}$-30.9%	$_{\text{CD}}$-54.3%
7	$_{\text{C}}$ 4.5 s	$_{\text{C}}$ 3.772	$_{\text{C}}$ 1.188	$_{\text{C}}$-36.8%	$_{\text{C}}$-64.6%	$_{\text{CD}}$ 0.6 s	$_{\text{CD}}$-37.2%	$_{\text{CD}}$-59.9%
8	$_{\text{C}}$ 5.1 s	$_{\text{C}}$ 3.974	$_{\text{C}}$ 1.070	$_{\text{C}}$-35.4%	$_{\text{C}}$-63.6%	$_{\text{CD}}$ 0.5 s	$_{\text{CD}}$-32.8%	$_{\text{CD}}$-53.9%
9	$_{\text{CT}}$ 10.2 s	$_{\text{CT}}$ 4.443	$_{\text{CT}}$ 1.273	$_{\text{CT}}$-31.1%	$_{\text{CT}}$-56.5%	$_{\text{CD}}$ 0.5 s	$_{\text{CD}}$-29.8%	$_{\text{CD}}$-53.5%
10	$_{\text{CT}}$ 10.2 s	$_{\text{CT}}$ 5.014	$_{\text{CT}}$ 1.435	$_{\text{CT}}$-36.8%	$_{\text{CT}}$-65.3%	$_{\text{CD}}$ 1.4 s	$_{\text{CD}}$-36.7%	$_{\text{CD}}$-61.1%
11	$_{\text{CT}}$ 10.3 s	$_{\text{CT}}$ 5.654	$_{\text{CT}}$ 1.620	$_{\text{CT}}$-36.7%	$_{\text{CT}}$-66.8%	$_{\text{CD}}$ 1.0 s	$_{\text{CD}}$-37.2%	$_{\text{CD}}$-63.9%
12	$_{\text{CT}}$ 10.3 s	$_{\text{CT}}$ 6.163	$_{\text{CT}}$ 1.912	$_{\text{CT}}$-31.5%	$_{\text{CT}}$-59.7%	$_{\text{CD}}$ 1.1 s	$_{\text{CD}}$-33.3%	$_{\text{CD}}$-57.6%
13	$_{\text{CT}}$ 10.3 s	$_{\text{CT}}$ 6.548	$_{\text{CT}}$ 2.135	$_{\text{CT}}$-32.2%	$_{\text{CT}}$-57.6%	$_{\text{CD}}$ 1.3 s	$_{\text{CD}}$-31.4%	$_{\text{CD}}$-55.3%
14	$_{\text{CT}}$ 10.3 s	$_{\text{CT}}$ 6.907	$_{\text{CT}}$ 2.287	$_{\text{CT}}$-40.4%	$_{\text{CT}}$-66.1%	$_{\text{CD}}$ 1.5 s	$_{\text{CD}}$-40.8%	$_{\text{CD}}$-64.8%
15	$_{\text{CT}}$ 10.4 s	$_{\text{CT}}$ 7.120	$_{\text{CT}}$ 2.247	$_{\text{CT}}$-38.9%	$_{\text{CT}}$-63.8%	$_{\text{CD}}$ 1.4 s	$_{\text{CD}}$-38.5%	$_{\text{CD}}$-63.4%
16	$_{\text{CT}}$ 10.4 s	$_{\text{CT}}$ 7.681	$_{\text{CT}}$ 2.422	$_{\text{CT}}$-37.4%	$_{\text{CT}}$-63.3%	$_{\text{CD}}$ 1.2 s	$_{\text{CD}}$-38.1%	$_{\text{CD}}$-61.7%

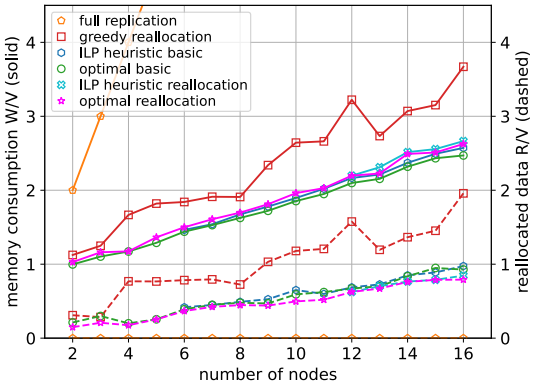
(b) Accounting workload: N=344, Q=4461, metadata.



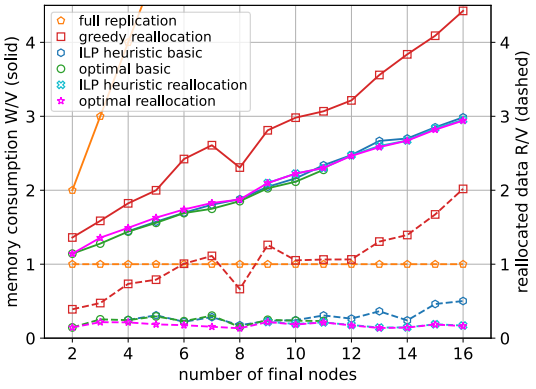
(a) Changing workload, TPC-H.



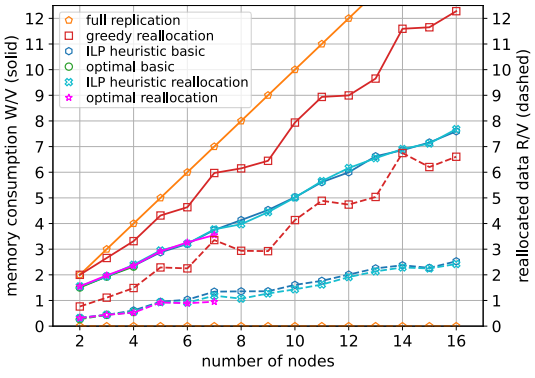
(b) Adding a node, TPC-H.



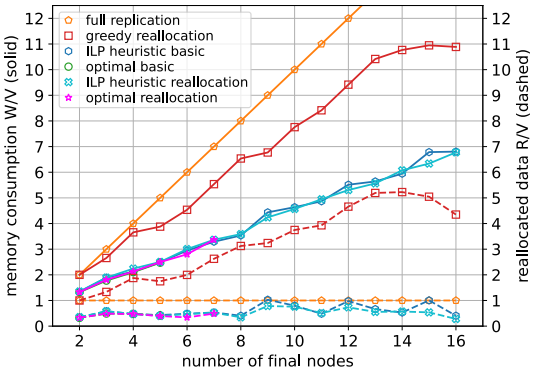
(c) Changing workload, TPC-DS.



(d) Adding a node, TPC-DS.

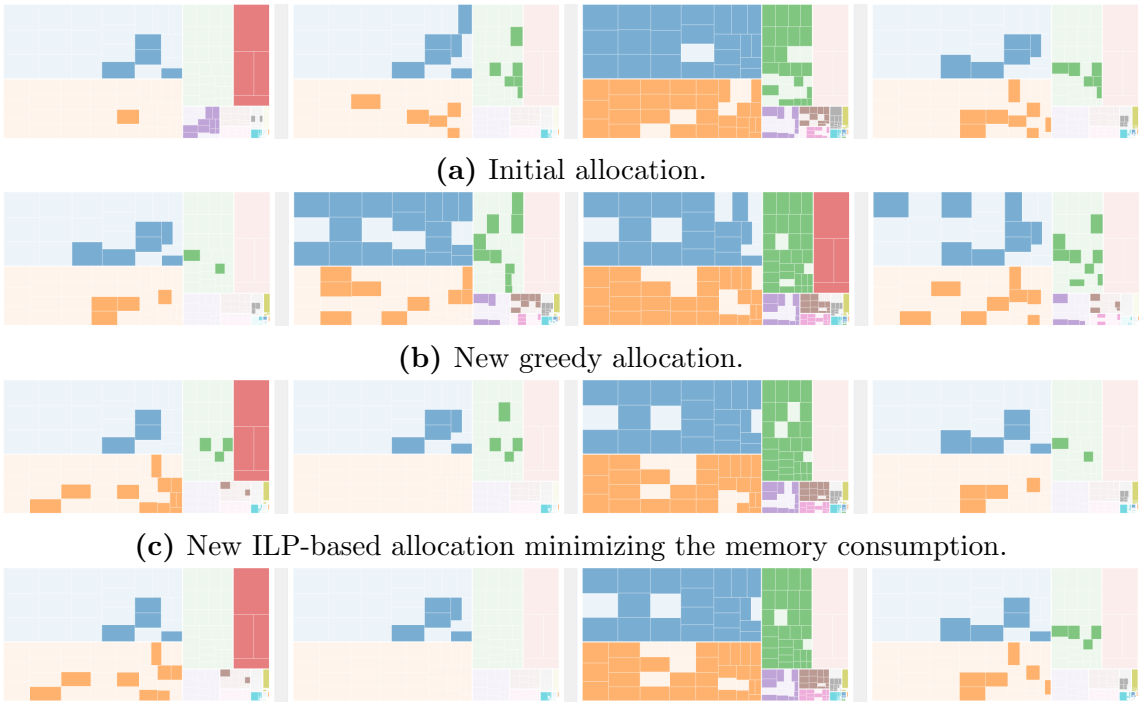


(e) Changing workl., accounting workload.



(f) Adding a node, accounting workload.

Figure 6.13.: Evaluation of data **reallocation approaches** for (i) changing workloads for a database cluster with 2 - 16 nodes and (ii) adding a node to a database cluster with 1 - 15 nodes, resulting in 2 - 16 final nodes: memory consumption and amount of reallocated data for full replication, the greedy approach, basic ILP-based approaches, and ILP-based reallocation approaches.



(a) Initial allocation.

(b) New greedy allocation.

(c) New ILP-based allocation minimizing the memory consumption.

(d) New ILP-based allocation minimizing the amount of reallocated data.

Figure 6.14.: Visualization of allocated data for changing workload: TPC-DS with $K = 4$ nodes.

Compared to the greedy approach, ILP-based approaches are always able to find better solutions with a lower memory consumption and less reallocated data. For TPC-H, ILP-based approaches require up to 16% less data (for $K = 3$, basic model) while reducing the amount of reallocated data by up to 41% (for $K = 3$, reallocation model). For the more complex TPC-DS and accounting workload, the gains are higher: the reallocation models produce allocations with a 7 - 32% (TPC-DS) and 22 - 41% (accounting workload) lower replication factor, and, thereby, also reduce the amount of reallocated data by up to 77% for TPC-DS (see $K = 4$, optimal) and up to 72% for the accounting workload (see $K = 7$, optimal). Recap, using the reallocation model, we can reliably minimize the amount of reallocated data in the case of changing workloads. The quality of our ILP-based allocations for complex workloads is underlined by the fact that, for the accounting workload, the memory consumption of the new allocations is close to the amount of reallocated data of the greedy approach (see Figure 6.13e)

Using our basic model, which solely focuses on memory, provides similar results and outperforms the greedy approach regarding memory consumption and reallocated data for almost all problem instances: Only TPC-DS with $K = 3$ nodes, the amount of reallocated data is larger than for the greedy approach. While the memory consumption of the basic model is slightly better than the reallocation model, the reallocation costs to achieve the new allocation become slightly worse.

We could calculate optimal reallocations faster than allocations for the basic problem because the allocation state $s_{i,k}, i = 1, ..., N, k = 1, ..., K$, may help the solver to limit the solution space with minimized reallocations. Nevertheless, the optimal reallocation model is not tractable for increasingly complex problem instances. Using ILP-based heuristics (see Table 6.19), we could calculate solutions for all problem instances in about 10 s or less while obtaining near-optimal (if comparable/tractable) results (see Figure 6.13c and 6.13e).

The visualized allocations in Figure 6.14 show that the new ILP-based allocations are similar to the initial allocation, while the greedy approach reallocates much data.

Adding a Node for Increasing Workload

Second, we evaluate reallocation approaches for changing cluster sizes, particularly when adding a node to the cluster. We assume an initial optimized allocation (calculated using our ILP-based approach) that enables an even load balancing of all TPC-H, TPC-DS, or accounting queries to partial replicas. We want to add an additional node to the cluster to scale the workload and calculate a new allocation while considering data reallocation costs. The ratio of the individual query frequencies does not change. Again, we evaluate approaches with regard to the memory consumption W and the amount of reallocated data R (see Equation (6.1)).

Table 6.20 and 6.21 compare optimal and heuristic ILP-based approaches with a calculation time of about 10 s or less to the greedy approach. Figure 6.13b, 6.13d, and 6.13f show all results including full replication. Figure 6.15 visualizes the initial allocation with 3 and the final allocations with 4 nodes for TPC-DS.

The results are similar to those of changing workloads: ILP-based approaches outperform the greedy approach, and ILP-based heuristic solutions can be calculated quickly. Again, the gains of ILP approaches over the greedy algorithm are greater for the more complex TPC-DS and accounting workload. Looking at the specific gains for the amount of reallocated data, we find that the advantages of ILP-based approaches are larger, e.g., reducing the TPC-H, TPC-DS, and accounting reallocation costs by up to 62% (TPC-H, $K = 9$, optimal), 92% (TPC-DS, $K = 16$, optimal), and 94% (accounting workload, $K = 16$, heuristic), respectively (while the gains for the memory consumption are similar). The higher savings are because the old and new workloads are similar, i.e., only the workload intensity changes. The optimal reallocation costs are small. However, the greedy approach recalculates the entire allocation to keep the overall memory consumption optimized. As a result, it has to reallocate much data. Note, the (normalized) amount of reallocated data for the greedy approach is even higher than 1, which corresponds to simply using a full replica for the increased load. In contrast, we see that the ILP-based approaches reallocate significantly less data than full replication when adding a node, particularly for the more complex TPC-DS and accounting workload.

Table 6.20.: Optimal solutions of data reallocation approaches for **adding a node** to a database cluster with 1 - 15 nodes, resulting in $K = 2$ - 16 final nodes: calculation time, memory consumption $\frac{W^R}{V}$ and amount of reallocated data $\frac{R^R}{V}$ for the ILP-based *reallocation* approach, and relative memory $\frac{W^R}{W^{GR}}$ and reallocation costs $\frac{R^R}{R^{GR}}$ savings compared to the greedy approach; for the ILP-based *basic* approach, calculation time, and relative memory $\frac{W^*}{W^{GR}}$ and reallocation costs $\frac{R^*}{R^{GR}}$ savings compared to the greedy approach.

K	time$_{W^R}$	$\frac{W^R}{V}$	$\frac{R^R}{V}$	$\frac{W^R}{W^{GR}}$	$\frac{R^R}{R^{GR}}$	time$_{W^*}$	$\frac{W^*}{W^{GR}}$	$\frac{R^*}{R^{GR}}$
2	0.0 s	1.406	0.491	-10.8%	-17.5%	0.1 s	-13.9%	-12.2%
3	0.0 s	1.710	0.420	-23.4%	-58.3%	0.2 s	-28.3%	-33.6%
4	0.1 s	1.974	0.406	2.8%	-39.0%	0.6 s	-7.6%	-11.6%
5	0.2 s	2.220	0.448	-16.6%	-57.7%	1.1 s	-19.3%	-53.7%
6	0.2 s	2.550	0.423	-9.0%	-48.7%	1.7 s	-12.1%	-44.0%
7	0.3 s	2.856	0.427	-6.6%	-52.5%	5.1 s	-12.3%	-44.9%
8	0.3 s	3.002	0.454	-12.7%	-52.0%	2.6 s	-13.9%	-43.6%
9	0.4 s	3.375	0.429	-10.8%	-61.7%	8.4 s	-12.0%	-48.0%
10	0.8 s	3.685	0.369	-9.3%	-56.4%	8.9 s	-14.3%	-36.3%
11	0.8 s	3.742	0.374	0.8%	-14.5%	9.9 s	-2.3%	-4.4%
12	1.9 s	4.070	0.559	-7.9%	-36.3%	57.5 s	-9.0%	-28.1%
13	1.0 s	4.371	0.351	-8.4%	-60.5%	48.3 s	-11.1%	-38.3%
14	1.6 s	4.534	0.389	-10.2%	-59.6%	326.9 s	-11.4%	-52.6%
15	5.1 s	4.830	0.439	-5.8%	-46.5%	349.6 s	-6.8%	-27.0%
16	1.5 s	5.290	0.633	-7.9%	-46.4%	327.4 s	-9.6%	-33.2%

(a) TPC-H: N=61, Q=22, commercial database system, scale factor 10.

K	time$_{W^R}$	$\frac{W^R}{V}$	$\frac{R^R}{V}$	$\frac{W^R}{W^{GR}}$	$\frac{R^R}{R^{GR}}$	time$_{W^*}$	$\frac{W^*}{W^{GR}}$	$\frac{R^*}{R^{GR}}$
2	0.1 s	1.142	0.145	-16.1%	-62.9%	0.8 s	-16.1%	-62.9%
3	0.2 s	1.358	0.216	-14.4%	-54.5%	2.7 s	-19.4%	-46.5%
4	0.6 s	1.491	0.213	-18.2%	-71.0%	41.8 s	-21.0%	-66.8%
5	2.0 s	1.629	0.187	-18.5%	-76.4%	87.0 s	-22.0%	-62.8%
6	5.1 s	1.740	0.175	-28.1%	-82.6%	131.8 s	-30.2%	-77.2%
7	2.7 s	1.827	0.154	-30.0%	-86.1%	227.7 s	-32.9%	-72.6%
8	3.1 s	1.875	0.133	-18.7%	-79.9%	343.9 s	-19.7%	-77.2%
9	14.5 s	2.098	0.218	-25.4%	-82.7%	2966.4 s	-27.9%	-80.3%
10	58.7 s	2.225	0.184	-25.4%	-82.5%	13227.5 s	-29.0%	-77.6%
11	58.9 s	2.302	0.213	-25.0%	-80.0%	20624.4 s	-25.8%	-78.3%
12	87.2 s	2.465	0.172	-23.3%	-83.8%	-	-	-
13	53.5 s	2.583	0.137	-27.4%	-89.5%	-	-	-
14	71.2 s	2.671	0.143	-30.4%	-89.7%	-	-	-
15	661.7 s	2.819	0.184	-31.1%	-89.0%	-	-	-
16	838.1 s	2.942	0.163	-33.5%	-91.9%	-	-	-

(b) TPC-DS: N=425, Q=95, commercial database system, scale factor 10.

K	time$_{W^R}$	$\frac{W^R}{V}$	$\frac{R^R}{V}$	$\frac{W^R}{W^{GR}}$	$\frac{R^R}{R^{GR}}$	time$_{W^*}$	$\frac{W^*}{W^{GR}}$	$\frac{R^*}{R^{GR}}$
2	12.3 s	1.329	0.329	-33.6%	-67.1%	22.1 s	-33.9%	-67.8%
3	366.2 s	1.808	0.485	-32.0%	-63.6%	2368.8 s	-33.2%	-63.3%
4	1128.8 s	2.128	0.473	-41.8%	-74.8%	835.8 s	-42.5%	-74.6%
5	9788.2 s	2.497	0.392	-35.6%	-77.6%	38661.1 s	-36.3%	-76.5%
6	2371.1 s	2.801	0.340	-38.2%	-83.0%	-	-	-
7	20141.0 s	3.369	0.487	-39.1%	-81.5%	-	-	-

(c) Accounting workload: N=344, Q=4461, metadata.

Table 6.21.: ILP-based heuristic solutions of data reallocation approaches for **adding a node** to a database cluster with 1 - 15 nodes, resulting in $K = 2$ - 16 final nodes: calculation time, memory consumption $\frac{W^R}{V}$ and amount of reallocated data $\frac{R^R}{V}$ for the ILP-based *reallocation* approach, and relative memory $\frac{W^R}{W^{GR}}$ and reallocation costs $\frac{R^R}{R^{GR}}$ savings compared to the greedy approach; for the ILP-based *basic* approach, calculation time, and relative memory $\frac{W}{W^{GR}}$ and reallocation costs $\frac{R}{R^{GR}}$ savings compared to the greedy approach. "$_{\text{D}}$" indicates a solution using the decomposition-based approach, "$_{\text{T}}$" indicates a solution using a time limit, "$_{\text{C}}$" indicates a solution using the clustering of workload-unintense queries on one node.

K	time$_{W^R}$	$\frac{W^R}{V}$	$\frac{R^R}{V}$	$\frac{W^R}{W^{GR}}$	$\frac{R^R}{R^{GR}}$	time$_W$	$\frac{W}{W^{GR}}$	$\frac{R}{R^{GR}}$
11	0.8 s	3.742	0.374	0.8%	-14.5%	$_{\text{D}}$ 1.0 s	$_{\text{D}}$-2.3%	$_{\text{D}}$-4.4%
12	1.9 s	4.070	0.559	-7.9%	-36.3%	$_{\text{D}}$ 0.6 s	$_{\text{D}}$-8.1%	$_{\text{D}}$-30.0%
13	1.0 s	4.371	0.351	-8.4%	-60.5%	$_{\text{D}}$ 0.8 s	$_{\text{D}}$-9.9%	$_{\text{D}}$-37.9%
14	1.6 s	4.534	0.389	-10.2%	-59.6%	$_{\text{D}}$ 0.5 s	$_{\text{D}}$-10.6%	$_{\text{D}}$-33.1%
15	5.1 s	4.830	0.439	-5.8%	-46.5%	$_{\text{D}}$ 0.9 s	$_{\text{D}}$-6.6%	$_{\text{D}}$-31.3%
16	1.5 s	5.290	0.633	-7.9%	-46.4%	$_{\text{D}}$ 0.6 s	$_{\text{D}}$-8.5%	$_{\text{D}}$-30.8%

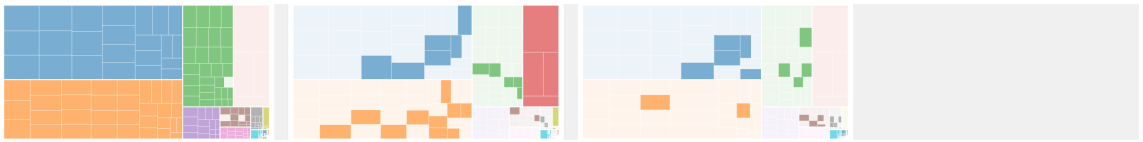
(a) TPC-H: N=61, Q=22, commercial database system, scale factor 10.

K	time$_{W^R}$	$\frac{W^R}{V}$	$\frac{R^R}{V}$	$\frac{W^R}{W^{GR}}$	$\frac{R^R}{R^{GR}}$	time$_W$	$\frac{W}{W^{GR}}$	$\frac{R}{R^{GR}}$
4	0.6 s	1.491	0.213	-18.2%	-71.0%	$_{\text{D}}$ 1.6 s	$_{\text{D}}$-20.7%	$_{\text{D}}$-66.3%
5	2.0 s	1.629	0.187	-18.5%	-76.4%	$_{\text{D}}$ 6.2 s	$_{\text{D}}$-21.3%	$_{\text{D}}$-61.0%
6	5.1 s	1.740	0.175	-28.1%	-82.6%	$_{\text{D}}$ 6.1 s	$_{\text{D}}$-30.0%	$_{\text{D}}$-78.0%
7	2.7 s	1.827	0.154	-30.0%	-86.1%	$_{\text{DT}}$ 5.8 s	$_{\text{DT}}$-30.8%	$_{\text{DT}}$-74.5%
8	3.1 s	1.875	0.133	-18.7%	-79.9%	$_{\text{DT}}$ 6.0 s	$_{\text{DT}}$-18.5%	$_{\text{DT}}$-73.7%
9	$_{\text{T}}$ 10.2 s	$_{\text{T}}$ 2.098	$_{\text{T}}$ 0.218	$_{\text{T}}$-25.4%	$_{\text{T}}$-82.7%	$_{\text{D}}$ 5.1 s	$_{\text{D}}$-27.2%	$_{\text{D}}$-81.5%
10	$_{\text{T}}$ 10.0 s	$_{\text{T}}$ 2.225	$_{\text{T}}$ 0.193	$_{\text{T}}$-25.4%	$_{\text{T}}$-81.7%	$_{\text{DT}}$ 8.4 s	$_{\text{DT}}$-27.5%	$_{\text{DT}}$-76.9%
11	$_{\text{T}}$ 10.1 s	$_{\text{T}}$ 2.300	$_{\text{T}}$ 0.213	$_{\text{T}}$-25.0%	$_{\text{T}}$-80.0%	$_{\text{DT}}$ 7.6 s	$_{\text{DT}}$-23.8%	$_{\text{DT}}$-71.1%
12	$_{\text{T}}$ 10.0 s	$_{\text{T}}$ 2.473	$_{\text{T}}$ 0.178	$_{\text{T}}$-23.1%	$_{\text{T}}$-83.3%	$_{\text{DT}}$ 8.0 s	$_{\text{DT}}$-23.0%	$_{\text{DT}}$-75.2%
13	$_{\text{T}}$ 10.1 s	$_{\text{T}}$ 2.598	$_{\text{T}}$ 0.143	$_{\text{T}}$-27.0%	$_{\text{T}}$-89.0%	$_{\text{DT}}$ 8.3 s	$_{\text{DT}}$-25.1%	$_{\text{DT}}$-72.2%
14	$_{\text{T}}$ 10.1 s	$_{\text{T}}$ 2.673	$_{\text{T}}$ 0.146	$_{\text{T}}$-30.3%	$_{\text{T}}$-89.6%	$_{\text{DT}}$ 10.3 s	$_{\text{DT}}$-29.7%	$_{\text{DT}}$-82.8%
15	$_{\text{T}}$ 10.2 s	$_{\text{T}}$ 2.838	$_{\text{T}}$ 0.188	$_{\text{T}}$-30.6%	$_{\text{T}}$-88.8%	$_{\text{DT}}$ 8.1 s	$_{\text{DT}}$-30.3%	$_{\text{DT}}$-72.4%
16	$_{\text{T}}$ 10.3 s	$_{\text{T}}$ 2.951	$_{\text{T}}$ 0.171	$_{\text{T}}$-33.3%	$_{\text{T}}$-91.6%	$_{\text{DT}}$ 10.3 s	$_{\text{DT}}$-32.5%	$_{\text{DT}}$-75.1%

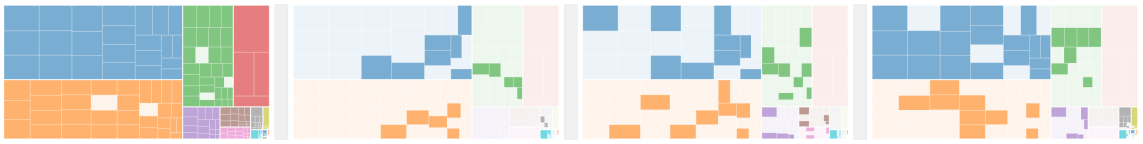
(b) TPC-DS: N=425, Q=95, commercial database system, scale factor 10.

K	time$_{W^R}$	$\frac{W^R}{V}$	$\frac{R^R}{V}$	$\frac{W^R}{W^{GR}}$	$\frac{R^R}{R^{GR}}$	time$_W$	$\frac{W}{W^{GR}}$	$\frac{R}{R^{GR}}$
2	$_{\text{C}}$ 0.1 s	$_{\text{C}}$ 1.350	$_{\text{C}}$ 0.350	$_{\text{C}}$-32.5%	$_{\text{C}}$-65.0%	$_{\text{C}}$ 0.2 s	$_{\text{C}}$-33.2%	$_{\text{C}}$-66.4%
3	$_{\text{C}}$ 0.6 s	$_{\text{C}}$ 1.891	$_{\text{C}}$ 0.581	$_{\text{C}}$-28.8%	$_{\text{C}}$-56.4%	$_{\text{C}}$ 0.5 s	$_{\text{C}}$-28.8%	$_{\text{C}}$-56.4%
4	$_{\text{C}}$ 0.8 s	$_{\text{C}}$ 2.242	$_{\text{C}}$ 0.473	$_{\text{C}}$-38.7%	$_{\text{C}}$-74.8%	$_{\text{C}}$ 1.4 s	$_{\text{C}}$-41.9%	$_{\text{C}}$-74.8%
5	$_{\text{C}}$ 1.8 s	$_{\text{C}}$ 2.502	$_{\text{C}}$ 0.397	$_{\text{C}}$-35.5%	$_{\text{C}}$-77.3%	$_{\text{C}}$ 3.2 s	$_{\text{C}}$-35.8%	$_{\text{C}}$-75.3%
6	$_{\text{C}}$ 3.8 s	$_{\text{C}}$ 3.003	$_{\text{C}}$ 0.470	$_{\text{C}}$-33.8%	$_{\text{C}}$-76.5%	$_{\text{CD}}$ 0.5 s	$_{\text{CD}}$-35.1%	$_{\text{CD}}$-75.9%
7	$_{\text{C}}$ 3.9 s	$_{\text{C}}$ 3.367	$_{\text{C}}$ 0.523	$_{\text{C}}$-39.2%	$_{\text{C}}$-80.1%	$_{\text{CD}}$ 1.2 s	$_{\text{CD}}$-40.6%	$_{\text{CD}}$-79.7%
8	$_{\text{C}}$ 3.6 s	$_{\text{C}}$ 3.588	$_{\text{C}}$ 0.343	$_{\text{C}}$-45.1%	$_{\text{C}}$-89.0%	$_{\text{CD}}$ 1.2 s	$_{\text{CD}}$-45.9%	$_{\text{CD}}$-87.0%
9	$_{\text{CT}}$ 10.3 s	$_{\text{CT}}$ 4.240	$_{\text{CT}}$ 0.785	$_{\text{CT}}$-37.4%	$_{\text{CT}}$-75.8%	$_{\text{CD}}$ 1.1 s	$_{\text{CD}}$-34.5%	$_{\text{CD}}$-68.2%
10	$_{\text{CT}}$ 10.3 s	$_{\text{CT}}$ 4.568	$_{\text{CT}}$ 0.758	$_{\text{CT}}$-41.1%	$_{\text{CT}}$-79.8%	$_{\text{CD}}$ 3.7 s	$_{\text{CD}}$-40.2%	$_{\text{CD}}$-78.7%
11	$_{\text{CT}}$ 10.4 s	$_{\text{CT}}$ 4.948	$_{\text{CT}}$ 0.512	$_{\text{CT}}$-41.2%	$_{\text{CT}}$-87.0%	$_{\text{CD}}$ 1.2 s	$_{\text{CD}}$-42.2%	$_{\text{CD}}$-87.7%
12	$_{\text{CT}}$ 10.4 s	$_{\text{CT}}$ 5.306	$_{\text{CT}}$ 0.733	$_{\text{CT}}$-43.6%	$_{\text{CT}}$-84.3%	$_{\text{CD}}$ 1.6 s	$_{\text{CD}}$-41.4%	$_{\text{CD}}$-79.1%
13	$_{\text{CT}}$ 10.4 s	$_{\text{CT}}$ 5.564	$_{\text{CT}}$ 0.554	$_{\text{CT}}$-46.6%	$_{\text{CT}}$-89.3%	$_{\text{CD}}$ 1.4 s	$_{\text{CD}}$-45.8%	$_{\text{CD}}$-87.2%
14	$_{\text{CT}}$ 10.5 s	$_{\text{CT}}$ 6.074	$_{\text{CT}}$ 0.569	$_{\text{CT}}$-43.6%	$_{\text{CT}}$-89.1%	$_{\text{CD}}$ 1.6 s	$_{\text{CD}}$-44.8%	$_{\text{CD}}$-89.9%
15	$_{\text{CT}}$ 10.5 s	$_{\text{CT}}$ 6.337	$_{\text{CT}}$ 0.537	$_{\text{CT}}$-42.1%	$_{\text{CT}}$-89.4%	$_{\text{CD}}$ 3.3 s	$_{\text{CD}}$-38.0%	$_{\text{CD}}$-80.1%
16	$_{\text{CT}}$ 10.6 s	$_{\text{CT}}$ 6.775	$_{\text{CT}}$ 0.280	$_{\text{CT}}$-37.8%	$_{\text{CT}}$-93.6%	$_{\text{CD}}$ 3.4 s	$_{\text{CD}}$-37.5%	$_{\text{CD}}$-90.7%

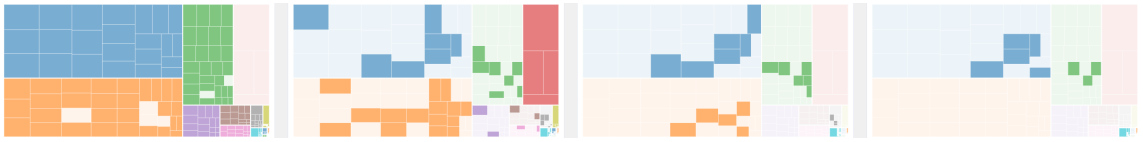
(c) Accounting workload: N=344, Q=4461, metadata.



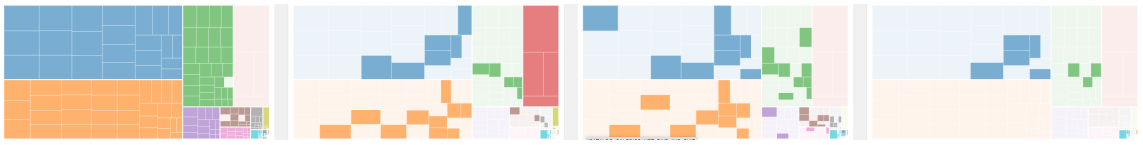
(a) Initial allocation.



(b) New greedy allocation.



(c) New ILP-based allocation minimizing the memory consumption.



(d) New ILP-based allocation minimizing the amount of reallocated data.

Figure 6.15.: Visualization of allocated data for adding a node: TPC-DS with $K = 4$ final nodes.

For TPC-DS, we see that the reallocation model can reduce the reallocation costs compared to the basic model for some problem instances noticeably while the memory consumption stays close to optimal (see Figure 6.13d, $K = 13, 15, 16$).

Again, the visualized allocations (see Figure 6.15) illustrate that the new ILP-based allocations are similar to the initial allocation, while the greedy approach reallocates much data. In particular, the added node has to allocate comparably little data when using the ILP-based approaches.

6.4. Summary

In this chapter, we evaluated our ILP-based allocation approaches for TPC-H, TPC-DS, and a real-world accounting workload. We calculated solutions for the basic read-only problem and all our extensions, considering node failures, workload uncertainty, data modification costs, and data reallocation costs. For all problem versions, we evaluated the tractability of optimal solutions and our ILP-based heuristics.

While optimal solutions are only tractable for smaller problem instances (depending on the workload, cluster size K, and the extension's complexity), we demonstrated that we could calculate ILP-based heuristic solutions quickly: Specifically, we were able to calculate heuristic solutions for all workloads, extensions, and clus-

ter sizes in under 100 s; most provided heuristic solutions took only a few seconds. It should be noted that we also included additional selected heuristic solutions with higher ($\geq$ 100 s) runtimes, e.g., when using fewer clustered queries or a decomposition with larger chunks; these solutions often provided better allocations in return. Overall, we demonstrated that the specific decomposition (i.e., chunk sizes), number of clustered queries, and solver relaxations could be chosen and combined flexibly, thereby balancing solution quality (i.e., memory consumption) and calculation time.

Our evaluations show that our ILP-based heuristics provide significantly better solutions than state-of-the-art approaches. We highlight selected top results in the following: For the accounting workload, we could almost halve the amount of allocated data (see Table 6.9b, $K = 8$). We could reduce the maximum node's load limit L in case of node failures by up to 45% (see Table 6.11b, $K = 14$) while using less data. Our solutions that consider multiple input scenarios provide a better combination of low memory consumption and high throughput against unseen workload scenarios (see Figure 6.9 - 6.11). For TPC-DS, we could reduce the modification costs by 55% while requiring 39% less data (see Table 6.17, $K = 11$). We could reduce the amount of reallocated data by up to 94% while the new allocation requires 38% less data (see Table 6.21c, $K = 16$).

6.5. Limitations of our Approaches and Evaluation

In this section, we discuss the limitations of our allocation approaches and the conducted evaluations. Our specific results depend on the input, e.g., the (i) accessed fragments per query and (ii) query costs:

The (i) accessed fragments per query depend on the chosen partitioning. We used individual columns as fragments, which are also used in Rabl and Jacobsen's paper [181]. Another option would be horizontal partitioning, which is also supported by our partition-agnostic (see Section 3.1.1) allocation models. To decide which fragments are possibly accessed by which query for a horizontally partitioned system, we can use rules of query containment research (e.g., [97]). However, the decision may depend on the knowledge of existing data dependencies between attributes. If we cannot guarantee that a row is not accessed by query j, its corresponding fragment has to be added to the accessed fragments q_j so that it is unambiguous to determine all replica nodes where the query can be executed.

For obtaining (ii) query costs and running end-to-end experiments, we used a commercial, columnar, in-memory database system in this thesis. Other database systems would naturally lead to differing query execution costs, fragment sizes (due to a different physical representation, also caused by encodings), and runtime behavior when processing queries. In our previous publications [102, 104, 195], we evaluated our approaches for PostgreSQL (with different query costs and fragment sizes) and obtained similar results.

For modeling query costs, we (and the greedy state-of-the-art approach [181]) use a single floating point parameter, which cannot account for the complex execution

behavior within database systems. In our end-to-end evaluations, we measured that actual query execution costs may differ. However, allocations can often compensate for these model inaccuracies because many queries are executable at multiple nodes, which can flexibly share the actual load. Further, our extension for workload uncertainty enables specifying multiple query workload shares, which are all considered for the solution. In practice, monitoring whether modeled input parameters are adequately accurate is advisable.

Inaccuracies of model inputs bring us to another limitation of this work. Workloads and data are dynamic. Hence, model inputs may change over time. While we have not implemented this situation for an end-to-end evaluation, we argue that our approach with its extensions (e.g., considering reallocation costs) can also be used in dynamic environments. Therefore, we have to monitor and determine the changing model input, quickly (see Section 5.2) calculate a corresponding reallocation (see Section 5.3.4), and conduct the actual reallocation.

A further aspect of our allocation model and evaluation is the focus on scaling the query throughput. Designing for and investigating query latencies are also essential. The overall idea of query-driven workload distribution would stay the same. However, query timings and dispatching gain importance because they may lead to temporarily overloaded nodes, causing high query latencies. To decrease the probability of situations with overloaded nodes, we can use our approaches for increasing the load balancing's flexibility/robustness, i.e., considering node failures (see Section 5.3.1) and workload uncertainty (see Section 5.3.2).

7

Applying Solution Concepts to the Index Selection Problem

> The value of an idea lies in the using of it.
>
> Thomas Edison

Mathematical programming is a flexible approach to address versatile allocation problems, including the index selection of a database. This chapter shows how we can apply our solution concepts (i.e., a model decomposition, heuristic input reduction, and robustness extensions) to index selection approaches based on mathematical programming. Section 7.1 describes the index selection problem. Section 7.2 gives a short overview of index selection approaches, including those based on mathematical programming. We present our ILP-based heuristic approaches to solve larger problem instances and risk-averse model extensions in Section 7.3. Following, we evaluate our approaches in Section 7.4. Section 7.5 concludes this chapter with a short summary.

7.1. Index Selection Problem

Indexes are database structures to speed up database workloads by enabling quick data retrievals without scanning the entire database. However, storing indexes requires memory. As storage may be a scarce resource (in particular for in-memory database systems) [237], there is typically a trade-off between improved performance and increased storage consumption. Further, indexes do not always accelerate query execution because inserts, updates, and deletes may require expensive index maintenance operations [88].

Because (real-world) workloads can consist of versatile queries on large schemas with many attributes and workloads may change over, the automatic selection of suitable indexes is an important optimization problem. However, choosing the set of indexes to minimize the workload costs while considering storage constraints is complex and challenging.

7.1.1. Formalized Problem Description

We consider the problem of choosing secondary indexes for a workload with the goal to minimize the overall workload costs, e.g., the execution time. In the following, we formalize the index selection problem and, therefore, define the basic vocabulary and our notation:

Workload. The workload consists of Q queries over a set of tables with overall N attributes. Each query j is characterized by a set of attributes $q_j \subseteq \{1, ..., N\}$, $j = 1, ..., Q$, that are accessed during query execution.

Index. An index i is characterized by an *ordered* set of attributes from $\{1, ..., N\}$. A single index cannot incorporate attributes of multiple tables but is only created on a single table. The number of index attributes is also called *index width*. The necessary storage/memory consumption for an index i is denoted by m_i.

Index Candidates and Index Selection. The set of indexes that are considered and evaluated by index selection algorithms are called *index candidates* and denoted by I. An *index selection* $I^* \subseteq I$ is the set of chosen indexes out of a candidate set.

Workload Costs. The costs for a query $j = 1, ..., Q$ depend on the index selection I^* and are denoted by $c_j(I^*)$. Note, a query j can be of various types and include, e.g., selections, joins, inserts, updates, deletes. The total workload costs C for an index selection I^* are defined by the sum of query costs c_j of all queries j, multiplied by their number of occurrences denoted by f_j, $j = 1, ..., Q$, i.e., $C(I^*) := \sum_{j=1,...,Q} f_j \cdot c_j(I^*)$.

Storage Constraint. We assume that the memory consumed by the selected indexes must not exceed a certain budget A. The total storage consumption of a selection I^* amounts to $M(I^*) := \sum_{i \in I^*} m_i$.

Index Selection Problem. Finally, the index selection problem can be defined by minimizing the workload costs $C(I^*)$ while not exceeding a certain storage budget $M(I^*) \leq A$. *Both* the index selection I^* as well as the index candidate set I determine the solution quality.

Note, there are other variations of the index selection problem, differing in optimization goals (e.g., fixed targeted workload costs) and constraints (e.g., limiting the number of selected indexes) [123].

7.1.2. Challenges

Finding the optimal set of indexes I^* for a given workload and dataset while considering constraints is challenging for various reasons [123, 194]:

Index Candidate Selection. The number of (multi-attribute) indexes is usually huge [123, 239]. It increases with the number of indexable attributes, the number of columns per index, and the number of considered index types, e.g., B-trees, hash maps, or bit maps. Usually (because of their large number and combination possibilities), index selection algorithms cannot consider all (potentially beneficial) indexes that could be generated from an arbitrary combination of attributes (from the same table) that are part of the workload. To reduce the index candidate set, many algorithms focus on (syntactically) relevant indexes that contain only columns that appear together in at least one query or limit the index width (i.e., indexes with many attributes are ignored). Overall, choosing index candidates from all potential indexes is an essential part of index selection algorithms [34, 42, 57].

Index Interaction. The benefit of an index (e.g., whether it is used for a query) depends on the presence of other indexes, known as index interaction. Thus, index benefits cannot be determined independently [198], which increases the complexity of choosing suitable index candidates and selecting the best indexes out of them significantly.

Index Benefit Evaluation. An efficient and accurate cost estimation of the workload with different indexes is challenging [42, 123]. Index selection algorithms must quantify the workload costs with a given index selection and the storage consumption of individual indexes. Repeatedly creating a large set of index combinations physically and executing queries is naturally too expensive. Therefore, most index selection algorithms only estimate index benefits, e.g., using the database system's optimizer and its cost model. Including the optimizer for the determination of query costs guarantees that indexes are really used during query execution [42].

To ease this task, some database systems support *hypothetical indexes* [42] (also called virtual indexes [222]), whose existence is *only simulated* to generate query plans and, thus, cost estimations as if the index would actually be physically present. Still, the resulting what-if optimizer calls require time-consuming query optimizations and may, thus, be the bottleneck for index selection approaches [173]. To reduce the number of what-if calls during the index selection, we can use caching and workload context information [123]. Further, one can speed up what-if calls by exploiting that optimal query plans for different index sets often have the same structure and only differ in the table access costs [173].

When using optimizer-based cost estimations as input for index selection algorithms, the solution quality naturally depends on the cost estimation accuracy. Cost estimations may be inaccurate [32, 63] because of cardinality misestimations [135]

and inaccurate cost models [233]. To mitigate estimation errors, better estimation techniques can be used, including the usage or improvement of histograms [110], sampling [72, 95, 235] and sketches [190], cardinalities of query executions [69, 164, 210], and query execution costs [63, 152]. Overall, optimizer-based cost estimations offer a reasonable combination of speed and accuracy [123].

Potential Future Workloads and Selection Robustness. Future workloads may be uncertain, and an index selection may have to optimize for multiple workload scenarios [194]. In this case, the selection can optimize the expected performance but may also include robustness considerations to mitigate the risk of a potentially poor selection for specific workload scenarios. Overall, we may want to target an, on average, (slightly) suboptimal selection that, in contrast, has a more stable/robust performance (e.g., having a lower variance) for all regarded potential workloads.

7.2. Index Selection Approaches

Diverse index selection algorithms have been proposed since the 1970s [145, 171]. In the following, we give an overview of index selection approaches. Then, we describe the *Extend* algorithm [196] as an exemplary greedy index selection approach, which we use in our evaluation. Finally, we discuss approaches based on integer linear programming (ILP).

7.2.1. Classification of Approaches

Index selection algorithms differ in their underlying solution approaches and complexity [123]: We can distinguish (i) *imperative* approaches that step-wisely adapt their index selection following a set of rules and (ii) (more) *declarative* approaches, e.g., using mathematical programming [36, 37, 57] or reinforcement learning [124, 129, 141, 191, 202]. Imperative approaches conceptually either start with an empty index selection and extend their selection successively (e.g., [1, 42, 196, 222]) or start with a large candidate set that is reduced continuously (e.g., [34, 229]). With regard to the complexity, we identified more [34, 43] or less [229] sophisticated candidate selections and transformations to adapt the current index selection.

There are also other dimensions to distinguish index selection algorithms [123], e.g., whether they are (as yet) pure academical proposals [124, 196] or related to commercial systems [1, 42, 222]. Especially the latter must consider further tuning aspects, such as stable/robust performance, scalability (i.e., suitability for also large problem instances), fast/time-bound selections, product integration, user interaction, and joint physical design tuning [1, 44, 184, 239].

We have comprehensively described and evaluated eight index selection algorithms along different dimensions, such as solution quality, runtime, multi-column support, solution granularity, and complexity [123]. Our corresponding implementation of the index selection evaluation platform is publicly available [122]. The evaluation

allowed us to assess strengths and weaknesses, and to infer insights for selecting indexes in general and each approach individually. We do not further discuss these aspects detailedly in this thesis. Instead, we focus, in the following, on index selection approaches based on ILP and the greedy *Extend* approach in greater detail.

7.2.2. The Greedy Extend Approach

The *Extend* approach of Schlosser et al. [196] is an imperative approach. The algorithm starts with an empty index selection and iteratively either adds or extends one index. The action is chosen greedily based on the highest ratio of cost benefit per storage.

Initially (given an empty index set), we choose the single-attribute index with the highest cost reduction per storage. Afterward, there are generally two options: either we add a new single-attribute index, or we extend an existing index by appending an attribute. The algorithm terminates if we reach the storage budget or we cannot further improve the selection.

This procedure accounts for index interaction as in each step, the effect of already chosen indexes is considered [123]. Originally, *Extend* does not limit the index width. But, it can be adapted to limit the index width [123] to solve more restricted index selection problems.

7.2.3. Approaches Based on Integer Linear Programming

Integer linear programming-based approaches guarantee optimal solutions but are generally not scalable. For practicable runtimes, we have to control the problem complexity. Naturally, the resulting problem simplifications might lead to suboptimal solutions for the unrestricted problem (see Section 4.2.2). Different ILP-based approaches differ in how they reduce the problem complexity.

Caprara et al. restrict the solution space by allowing only a single index per query [36, 37]. Thus, query accelerations with simultaneously applied indexes are not taken into account.

In contrast, Dash et al. consider multiple indexes per query (see also [172]) and multiple query plans that the optimizer can choose depending on existing indexes [57]. Naturally, the possibility of multiple query plans increases the problem complexity. This is why, we have to limit the number of index candidates (e.g., by limiting the index width) or admitted plans.

In the following, we describe our ILP formulation for the index selection problem [123]. We consider different index combinations/options K, which can be applied to queries. Each of these index combinations is a subset of index candidates, i.e., $k \subseteq I$, $k \in K$. The empty combination $k = \{\}$ is one of the considered combinations (i.e., $\{\} \in K$) and describes the option that no index is applied to a query. The costs of a query j using the index combination k are denoted by $c_{j,k}$. Recap, query

frequencies are denoted by $f_j, j = 1, ..., Q$; we use m_i for the storage consumption of an index $i \in I$, and A for the index storage budget (see Section 7.1.1).

To describe an index selection $I^* \subseteq I$, we use the following decision variables:

- $x_i \in \{0, 1\}$, $i \in I$, are allowed to be zero or one, indicating whether an index i is selected (1) or not (0).
- $y_k \in \{0, 1\}$, $k \in K$, are allowed to be zero or one, indicating whether an index combination k is applicable (1) or not (0). The values y_k can be derived by the selected indexes $x_i, k \in K$, $i \in I$.
- $z_{k,j} \in \{0, 1\}$, $k \in K$, $j = 1, ..., Q$, are allowed to be zero or one, indicating whether an index combination k is actually used (1) for query j or not (0).

The variables x, y, and z must be chosen such that the objective

$$\underset{x_i, y_k, z_{k,j} \in \{0,1\}, i \in I, j = 1, ..., Q, k \in K}{minimize} \sum\nolimits_{j=1,...,Q, k \in K} f_j \cdot c_{j,k} \cdot z_{k,j} \tag{7.1}$$

is minimized and the following (families of) constraints are satisfied:

$$\sum\nolimits_{k \in K} z_{k,j} = 1, \quad j = 1, ..., Q \tag{7.2}$$

$$\sum\nolimits_{i \in k} x_i \geq |k| \cdot y_k, \quad k \in K \tag{7.3}$$

$$z_{k,j} \leq y_k, \quad j = 1, ..., Q, k \in K \tag{7.4}$$

$$\sum\nolimits_{i \in I} m_i \cdot x_i \leq A. \tag{7.5}$$

The objective (7.1) minimizes the overall workload costs, which are the sum of all query costs with the used index combinations. The family of constraints (7.2) guarantees that a unique index combination/option k is used for each query j. The constraints (7.3) ensure that an index combination k is only applicable if all its indexes are selected. Note, the empty combination is always applicable (as $|\{\}| = 0$ always allows $y_{\{\}} = 1$), i.e., each query can be executed without indexes. The constraints (7.4) guarantee that an index combination k is only used for a query if it is applicable. The constraint (7.5) ensures that the index sizes m_i of all selected indexes do not exceed the memory budget A.

Approaches like (7.1) - (7.5) guarantee optimality but do not scale because the problem complexity strongly increases in the number of queries Q and index combinations $|K|$ (and, thus, index candidates $|I|$). Hence, solver-based approaches are either not applicable to larger problem instances or potentially lead to suboptimal selections due to reduced candidate sets [196].

Further, the ILP formulation (7.1) - (7.5) requires all cost coefficients $c_{j,k}$. Hence, another disadvantage of solver-based index selection approaches may be the high number of required cost estimations for different index combinations [57, 123, 173].

7.3. Heuristic and Risk-Averse Index Selection Using Mathematical Programming

ILP formulations guarantee optimal solutions but may require too high runtimes, especially when considering large index candidate sets or more complex risk-averse selections over multiple potential future workload scenarios. In Section 7.3.1 and 7.3.2, we present two approaches for heuristically selecting indexes while still leveraging the power of mathematical solvers. Following, we explain how we can obtain robust index selections by considering the performance variance of potential workloads in Section 7.3.3.

7.3.1. Hybrid Approach

ILP models like (7.1) - (7.5) guarantee optimal solutions for the given input. However, for larger workloads, the corresponding problem instances become quickly too large to solve optimally for two reasons: First, with an increasing number of queries Q, indexes $|I|$, and index combinations $|K|$, the ILP solver's calculation time becomes too high. Second, the ILP formulation (7.1) - (7.5) requires all cost coefficients $c_{j,k}$, which may require costly estimations.

For this reason, common ILP-based approaches limit the index width or the number of indexes per query. However, for complex workloads, broad indexes and index combinations with many indexes may be beneficial [123].

Our previous evaluation of index selection algorithms has shown that many existing approaches consider wide indexes and combinations with many indexes using a reasonable number of cost estimates. As current ILP solvers can often quickly calculate problem instances with thousands of index combinations, our idea is to use the estimates that are conducted while running existing heuristic approaches as input for the ILP model (7.1) - (7.5). Thereby, we could also preprocess or combine the input of multiple heuristics or different settings. In this process, our approach can improve the index selection by (i) a better selection of indexes based on thousands of cost estimates and (ii) a larger number of cost estimates.

The conducted number of cost estimates of a single heuristic is often small enough to quickly solve the resulting ILP problem optimally. However, with an increasing number of cost estimates (e.g., when using an increasing number of heuristics as input), selecting optimal indexes in a single step (i.e., using a single ILP program) may take practically too long. Following, we propose a general decomposition-based heuristic for selecting indexes using ILP.

7.3.2. Decomposition-Based Heuristic

A major problem of index selections is the potentially huge number of index candidates and, thus, combinations. However, limiting this number strongly upfront may lead to a poor overall selection. As the selection of optimal indexes in a single

step (i.e., using a single ILP program) may take practically too long, we propose to heuristically decompose the problem. In the following, we first describe our three-step approach for the index selection problem. Afterward, we discuss the approach and tuning possibilities.

Three-Step Approach for the Index Selection Problem

- **Step 1: Dividing Index Combinations into Subsets.** We divide all index combinations into smaller subsets, which may or may not (see [194]) overlap. By diving all index combinations instead of all index candidates, we ensure that each combination is part of at least one subset. The number of subsets (called chunks), the chunk size (i.e., the number of index combinations per chunk), and the allocation of index combinations to chunks are tunable.

  The option to apply no index is part of all chunks so that we (i) ensure not violating the budget because we can always select no index and (ii) do not select useless index combinations that are not better than applying no index. In addition, we could optionally add the option to apply single-attribute indexes for all chunks so that we do not select useless multi-attribute index combinations that are not better than applying the single-attribute index with the same leading attribute.

- **Step 2: Solving the Index Selection for Subsets of Combinations.** We solve the index selection problem (7.1) - (7.5) for all chunks independently and note the selected index combinations. The applied memory budget is tunable, e.g., identical with the overall memory budget A (see [194]).

- **Step 3: Solving the Index Selection for the Selected Index Combinations in Step 2.** We solve the index selection problem (7.1) - (7.5) with the union over all noted index combinations per chunk.

The key ideas of step 1 and step 2 are solving the index selection problem with smaller inputs (i.e., combination subsets) and excluding index combinations that are dominated by others. In step 3, the best selection of the best index combinations of the individual subsets is calculated.

The subproblems' complexity in step 2 can be controlled by choosing the chunks' sizes. Larger chunk sizes increase the selection complexity for individual chunks, but more index interaction can be taken into account. Further, the complexity of step 3 increases with the number of chunks because we select one combination for each chunk and query for the final selection. The applied memory budget A in step 2 allows for tuning what kind of index combinations are selected for step 3. Larger budgets may lead to an increasing number of memory-intensive index combinations, of which only a few can be selected in the final step 3. Too small budgets may filter out promising index combinations and, thus, lead to worse final results.

In case the candidate set after step 2 is too large, we could also repeatedly apply step 1 and 2 to reduce the number of index combinations [194].

The decomposition approach (step 1 - 3) is a general version of our previous proposal [194], for which we assumed a single index per (sub) query and disjoint candidate subsets in step 1. Assuming a single index per query makes a partitioning of index candidates into disjoint subsets a reasonable choice because we cannot miss opportunities that only arise when multiple indexes exist simultaneously. Further, non-overlapping chunks lead to non-overlapping selections in step 2 and, thus, potentially more diverse candidates for the final selection in step 3. In contrast, overlapping chunks better allow for considering index interactions when multiple indexes are applied simultaneously to a query. To assign indexes to chunks in step 1, we can use workload information. For example, by collecting *similar* indexes within chunks, we could early address index interaction more effectively [194].

7.3.3. Risk-Averse Index Selection for Multiple Workloads

Future workloads may be uncertain and be modeled as a set of potential workload scenarios. We consider S potential workload scenarios with probabilities $P_s, s = 1, ..., S$ and $\sum_{s=1,...,S} P_s = 1$. A workload scenario s is characterized by a set of queries j, which occur with given frequencies $f_{j,s}, j = 1, ..., Q, s = 1, ..., S$, within a certain time frame. Regarding a suitable index selection, such scenarios allow for optimizing the expected performance (risk-neutral) or more robust (risk-averse) objectives, which aim for a more stable performance across the individual scenarios. For optimizing the expected performance for multiple workload scenarios, we only have to adapt the optimization function (7.1) as follows:

$$\begin{gathered} minimize \\ x_i, y_k, z_{k,j} \in \{0,1\}, i = 1 \in I, j = 1, ..., Q, k \in K, s = 1, ..., S \end{gathered}$$

$$\sum\nolimits_{s=1,...,S} P_s \cdot \sum\nolimits_{j=1,...,Q,k \in K} f_{j,s} \cdot c_{j,k} \cdot z_{k,j} \tag{7.6}$$

Optimizing the expected performance for multiple workload scenarios as above does not increase the complexity of our ILP formulation because we can compute the expected frequencies upfront.

However, optimizing the expected/average performance may lead to widely varying performances for individual scenarios. For example, single scenarios could have a poor performance and still not benefit from indexes at all. Of course, other scenarios would, in contrast, benefit from indexes strongly. But in practice, a stable performance and avoiding poor performances for individual scenarios may be important.

One approach to achieve a more stable performance is considering the variance (of workload costs for the individual scenarios), called mean-variance optimization (MVO) [194]: MVO can balance the expected performance and performance deviations. Based on the workload costs for all individual scenarios

$$C_s(\vec{z}) = \sum\nolimits_{j=1,...,Q,k \in K} f_{j,s} \cdot c_{j,k} \cdot z_{k,j}, s = 1, ..., S, \tag{7.7}$$

the *expected workload costs* of an index selection (determined by $\vec{z}$) are given by

$$EC(\vec{z}) = \sum\nolimits_{s=1,\ldots,S} P_s \cdot C_s(\vec{z}) \tag{7.8}$$

and the associated *variance* of workload costs amounts to

$$VC(\vec{z}) = \sum\nolimits_{s=1,\ldots,S} P_s \cdot (C_s(\vec{z}) - EC(\vec{z}))^2. \tag{7.9}$$

The trade-off between expected costs EC and the variance VC can be modeled by the extended objective

$$\text{minimize}_{\vec{x},\vec{y},\vec{z}} \quad MVO(\vec{z}) = EC(\vec{z}) + \alpha \cdot VC(\vec{z}) \tag{7.10}$$

using a penalty parameter $\alpha \geq 0$. The penalty term in the objective provides an incentive to *avoid large deviations* in potential workload costs by selecting indexes such that queries of comparably heavy workload scenarios are sped up [194]. The decision variables and the constraints (7.2) - (7.5) remain *unchanged*. For $\alpha = 0$, we obtain the *linear* risk-neutral model (7.6) subject to (7.2) - (7.5). For $\alpha > 0$, we have a (more complex) binary *quadratic* problem (BQP), which still can be solved using standard solvers as long as the problem is sufficiently small. Recap, a suitable problem complexity can be achieved by applying our heuristic approaches to the BQP formulation (7.10) subject to (7.2) - (7.5).

MVO aims for a stable performance across all scenarios. Unfortunately, it, thereby, also avoids particularly cheap individual scenario costs. Therefore, we do not solely optimize the variance but also the expected performance. Another possibility to increase robustness but not penalize cheap scenarios is optimizing the index benefit per scenario using monotonically increasing concave functions, e.g., a logarithmic or exponential utility function [19, 197]. Further, one could also take the worst-case performance over all scenarios into account by using a penalty approach [197, 228]. In the following, we do not further focus on MVO alternatives.

7.4. Evaluation

In this section, we evaluate index selection algorithms for analytical TPC-H and TPC-DS workloads. After describing the experimental setup (Section 7.4.1), we first compare the greedy *Extend* [196] and our basic optimal ILP-based algorithm regarding the quality of the identified solutions and their corresponding calculation time for different storage budgets. Afterward, we evaluate our hybrid and decomposition ILP heuristics in Section 7.4.3. Finally, we show the effect of our risk-averse approach to optimize the cost variance of multiple workload scenarios (see Section 7.4.4).

7.4.1. Experimental Setup and Determining Model Inputs

All index selection experiments were executed on an Apple M1 Pro CPU with 8 cores and 32 GB RAM. We chose PostgreSQL as database system for the evaluation because it exposes an interface to retrieve cost estimates for hypothetical indexes, which are the basis for many (including our evaluated) index selection algorithms. To obtain the model inputs, we set up a PostgreSQL 14.5 database system with the extension HypoPG [189] (version 1.3.1). HypoPG enables the creation, deletion, and size estimation of hypothetical indexes. Using PostgreSQL's `EXPLAIN` command, we can inspect query plans with arbitrary hypothetical index selections. Thereby, we can determine the used indexes and the plan's estimated total execution costs, which we use as model inputs $c_{j,k}, j = 1, ..., Q, k \in K$, i.e., the costs for a query j using index combination k. Naturally, the query cost estimates may be inaccurate (see Section 7.1.2), but they provide a consistent and reasonable input for comparing the algorithms [123].

We loaded TPC-H and TPC-DS tables with scale factor 10. We used PostgreSQL's `ANALYZE` command to ensure up-to-date statistics, which are used for query cost estimations and for storage consumption predictions of indexes.

As in our previous evaluation [123], we excluded the queries 2, 17, and 20 for TPC-H and the queries 4, 6, 9, 10, 11, 32, 35, 41, and 95 for TPC-DS because their estimated costs in PostgreSQL were orders of magnitude higher than those of the other queries: Without the exclusion, these queries would dominate the costs of the entire workload. Hence, the index selection would be less complex because an index that decreases the costs of at least one of these queries would always significantly outperform indexes for other queries [123].

We evaluate index selection algorithms for read-only workloads. Thus, index maintenance costs are not taken into account. Furthermore, we used PostgreSQL's default index type, a non-covering B-tree, which is supported by HypoPG.

The quality of an index selection algorithm depends on the specific evaluation scenario, e.g., the model input, constraints, and algorithm's configuration [123]. Compared to our previous evaluation [123], our goal is not a broad evaluation of many algorithms. Rather, we focus on a deeper evaluation of ILP-based index selection algorithms for different problem settings (given by the considered index combinations/options) to outline (i) their strengths and weaknesses in greater detail and (ii) the effect of our heuristic ILP approaches. For a comparison to solutions of state-of-the-art approaches, we selected the greedy *Extend* [196] algorithm because its solutions were among the bests for different workloads and storage budgets [123]. We use the *Extend* implementation from our previous evaluation [123]. We implemented the ILP models in AMPL [78] and used Gurobi 9.5.2 [94] as solver. In this context, we extended our open-source evaluation platform [122] with the capability to export the conducted cost estimates and solve ILP models heuristically with our decomposition approach.

7.4.2. Risk-Neutral Optimal Approach

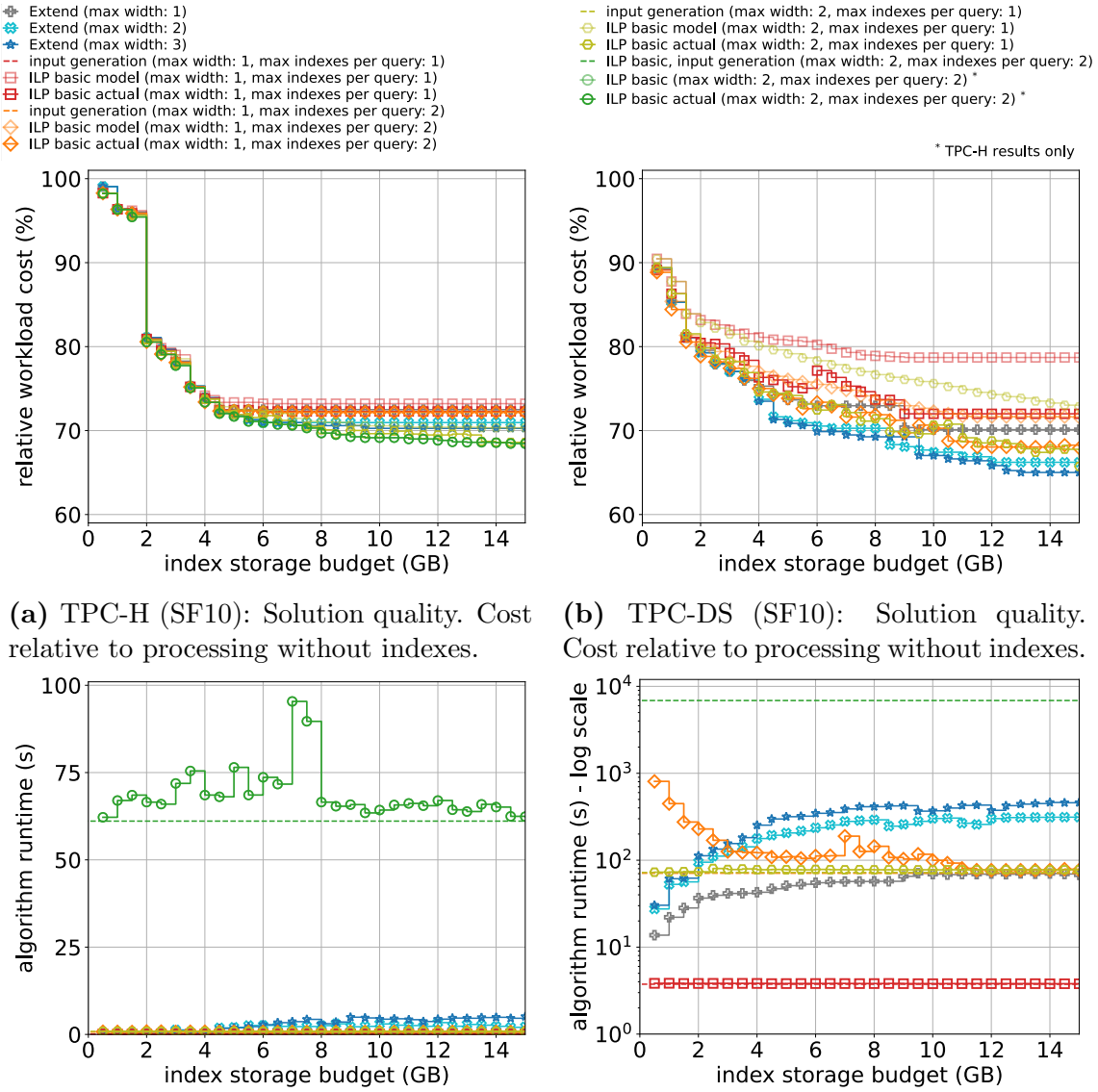
In this section, we compare the greedy *Extend* [196] algorithm and our basic optimal ILP-based algorithm (see (7.1) - (7.5)) for different index storage budgets ranging from 500 MB to 15 GB and different algorithm settings that restrict the index width (for *Extend* and the ILP formulation) and maximum number of applied indexes per query for the ILP formulation. Recap, *Extend* does not limit the number of indexes per query by its operating principle (see Section 7.2.2). For ILP-based solutions, we have to restrict the index width and number of index combinations to limit the number of (costly) what-if calls and the ILP model's complexity.

For all ILP solutions, we first show the results as reported by the solver, which are limited by the number of indexes per query. Because the solver guarantees optimal solutions for the restricted problem, these solutions become better with an increasing budget. Further, we apply and evaluate the solver's solutions to obtain the corresponding actual results, which are not limited by the number of indexes per query anymore. Compared to the results of the restricted problem, the actual results may, thus, be better. Note, because these improvements are not considered by the solver, solutions for larger budgets do not have to become better. Besides the solution quality (i.e., the relative workload cost compared to the processing costs without indexes), we report the algorithms' runtime. For ILP-based approaches, we also state the time share of the input generation, including all what-if estimations.

The complexity and, thus, runtime of *Extend* and especially the ILP-based approach increases with the number of queries, indexes, and combinations. For all experiments, we consider (syntactically) relevant indexes and combinations that contain only columns and indexes that appear together in at least one query. In particular for the more complex TPC-DS workload, considering only relevant combinations reduces the runtime significantly compared to a full enumeration. ILP-based approaches require all model inputs by means of relatively expensive what-if optimizer calls for these indexes. As a result, for more complex settings, the number of what-if calls and, thus, runtime becomes eventually too large. For this reason, we limit (i) the index width for ILP-based approaches to 2, and (ii) the maximum number of indexes per query (for the what-if calls) to 2. Further, with an increasing number of indexes and index combinations, the solver time increases heavily. Consequently, the optimal ILP approach with complex settings (e.g., with an index width of 2 and 2 indexes per query) was not solvable within 8 h for the larger TPC-DS workload anymore. Thus, TPC-DS results for the ILP-based approach with these settings are not presented but only the time for the input generation. We evaluate the greedy *Extend* approach up to an index width of 3. Figure 7.1 shows the result.

TPC-H Results

For TPC-H, workload costs (see Figure 7.1a) for different algorithms and settings are similar up to an index storage budget of 4 GB. For larger budgets, more complex settings, i.e., (greedy and ILP) approaches with a maximum index width of 2 and



(a) TPC-H (SF10): Solution quality. Cost relative to processing without indexes.

(b) TPC-DS (SF10): Solution quality. Cost relative to processing without indexes.

(c) TPC-H (SF 10): Observed runtime.

(d) TPC-DS (SF 10): Observed runtime.

Figure 7.1.: Index selection results for the greedy *Extend* and the basic optimal ILP-based approach with different settings.

3, find better index selections. For budgets over 7.5 GB, the ILP approach with a maximum index width of 2 and up to 2 indexes per query finds the best solutions. Note, despite potentially more indexes per query and a greater maximum index width of 3, *Extend* could not find better solutions.

Comparing the ILP model and actual results, we find that they are mostly close to each other. Only for the model with an index width of 2 and 1 index per query, the actual results deviate more strongly (for a storage budget of 8.5, 13.5, 14, 14.5, and 15 GB): As the ILP solutions of the model with index width 2 and 2 indexes

per query show, there are beneficial combinations that are, in this case, missed by the simpler model but whose effects occur in the actual results.

Regarding the calculation time, the best ILP solutions come with the price of the highest runtimes with up to 95 s, while each *Extend* solution was obtained in under 6 s (see Figure 7.1c). The major part (i.e., 61 s) of the ILP runtime was required for the model's input generation.

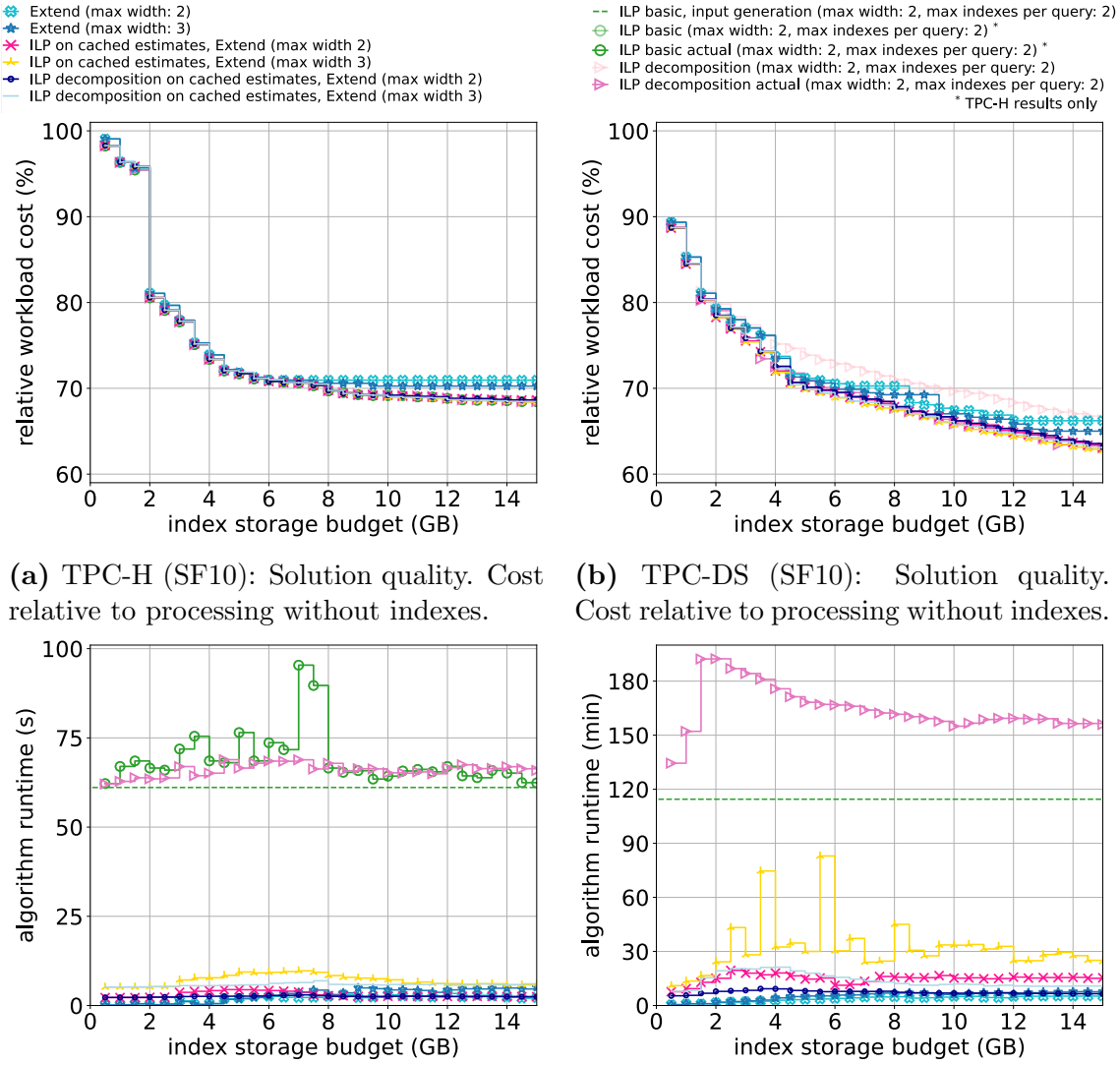
TPC-DS Results

For the more complex TPC-DS workload, the cost results (see Figure 7.1b) are more diverse for the different algorithms and settings. Index selections with wider indexes or multiple indexes per query are already noticeably better for budgets of 1 GB. For larger budgets over 4 GB, the best solutions contain combinations with wide and more than two indexes per query, which would practically take too long to calculate using the optimal ILP approach. Thus, the greedy *Extend* algorithm is better suitable for complex index selection problems compared to the optimal ILP-based approach. Comparing the ILP model and actual results, we find that they significantly deviate from each other. The reason for this is that using multiple indexes per query is highly beneficial for TPC-DS. These (beneficial) multi-index combinations are not considered by simpler models, but their effects are, then, noticeable in the actual results.

Summary: *Simple index selection formulations with a limited index width and maximum number of indexes per query can be solved optimally using ILP. For the simpler TPC-H workload, these (limited) ILP formulations could find better solutions than the greedy Extend approach. However, for the more complex TPC-DS workload and larger budgets, the best (found) solutions consist of index combinations with wide indexes and require multiple indexes per query, which was not tractable using an optimal ILP approach anymore. In this case, the greedy Extend approach [196] provided the best results.*

7.4.3. Risk-Neutral Heuristic Approaches

In this section, we evaluate our hybrid and decomposition ILP heuristics (see Section 7.3.2) for selecting indexes. On the one hand, we evaluate our hybrid heuristic based on the conducted cost estimates of the greedy *Extend* approach with an index width of 2 and 3. For our hybrid heuristic, we use *Extend*'s cached estimates for the largest budget (15 GB), which contains the most entries. On the other hand, we evaluate how the decomposition heuristic influences the solution quality (i.e., the solution's overall workload costs) and runtime compared to the optimal ILP formulation. In this context, we also evaluate more complex index selection formulations with wider and more indexes per query for TPC-DS, for which the optimal solution



(a) TPC-H (SF10): Solution quality. Cost relative to processing without indexes.

(b) TPC-DS (SF10): Solution quality. Cost relative to processing without indexes.

(c) TPC-H (SF 10): Observed runtime.

(d) TPC-DS (SF 10): Observed runtime.

Figure 7.2.: Index selection results for the greedy *Extend* and ILP-based decomposition approach with different settings.

took practically too long (see Section 7.4.2). When using the decomposition approach, we included the option to apply all single-index combinations to all chunks. Figure 7.2 shows the result.

TPC-H Results

Our hybrid approach, applying ILP on the cost estimates of the greedy *Extend* approach, improves the index selection noticeably for TPC-H with storage budgets over 7.5 GB (see Figure 7.2a). The corresponding results were obtained in under 10 s (see Figure 7.2c). Applying our decomposition approach (using 2 chunks) on

top of the hybrid approach could decrease the maximum overall runtime to under 7 s without affecting the solution quality noticeably.

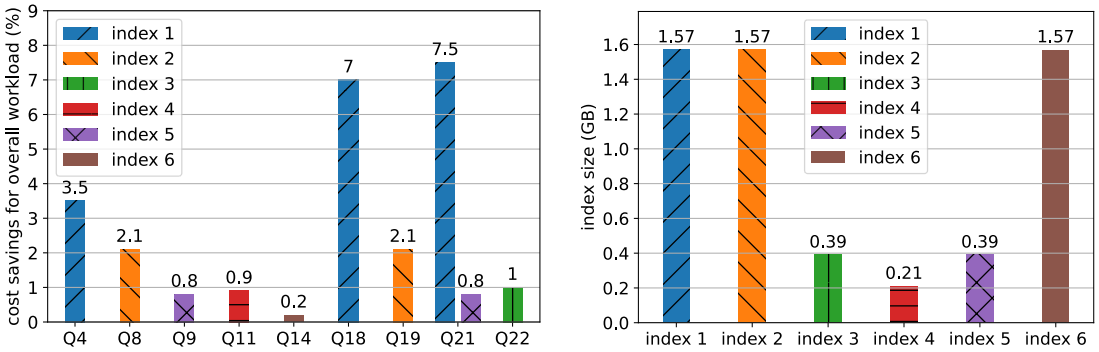
Following, we compare the decomposition approach (using 5 chunks) against the optimal solution for TPC-H with an index width of 2 and 2 indexes per query. The option to apply no index and 92 *fixed combinations* with a single index per query are considered for each chunk. The decomposition heuristic provides the same solution quality while the solver's runtime is, in some cases, significantly faster, e.g., 8 s vs. 34 s for a budget of 7 GB. The time of the input generation is not influenced and took 61 s. The decomposition heuristic calculates $6 = 5 + 1$ ILP model instances (one for each of the five chunks and one to determine the final solution), which took at most overall 8 s (for a budget of 7 GB). For some budgets (e.g., 12.5, 13, 14.5, 15 GB), the decomposition heuristic took longer than the optimal solution, but it provided more stable calculation times.

TPC-DS Results

As for TPC-H, our hybrid heuristic also improves the index selection compared to the greedy *Extend* approach for TPC-DS. In contrast to TPC-H, we see improvements for all storage budgets (see Figure 7.2b). While the maximum runtime of *Extend* with an index width of 2 and 3 is about 5 and 8 minutes, respectively, the ILP solver's calculation time for our hybrid heuristic accounts for up to 14 (index width 2) and 75 (index width 3) minutes on top (see Figure 7.2d). Using our decomposition approach on top, we could reduce the maximum solver time from 14 to 4 minutes for an index width of 2 and from 75 to 14 minutes for an index width of 3 while barely affecting the solution quality.

Following, we evaluate the decomposition heuristic for an index width of 2 and 2 indexes per query using 100 chunks (with 1 no-index + 1065 single-index *fixed combinations*). Recap, we were not able to practically calculate solutions for an index width of 2 and 2 indexes per query with the optimal model. Using our decomposition heuristic, it is possible, although the required time for the input generation (almost 2 hours) and calculating the ILP model (20 - 79 minutes) remains overall higher than for the greedy *Extend* and hybrid approach. In return for the high calculation time, the solutions' quality of the decomposition approach is better than those of the greedy approach. However, the hybrid approach outperforms the pure decomposition heuristic for most budgets. Only for a budget of 3.5 and 13.5 GB, the pure decomposition heuristic provides slightly better results. We found (for example comparing the model and actual ILP decomposition results) that the best solutions benefit from applying more than two indexes per query.

Summary: *Our hybrid ILP heuristic can improve the index selection over the greedy Extend approach for TPC-H and TPC-DS. Our decomposition heuristic enables to reduce the solver time while we barely noticed a drop in the solution quality in*



(a) Index cost savings per query.

(b) Index sizes.

Figure 7.3.: TPC-H (SF10): Visualized ILP input (max width:1, max indexes per query: 1) for indexes with more than 0.1% cost savings based on the overall workload costs.

our experiments. When reducing the calculation time, we can also solve more complex index selection formulations with wider indexes and more indexes per query. As a result, our pure ILP decomposition approach could find a better index selection than the greedy Extend approach for the more complex TPC-DS workload. However, with an increasing number of indexes and indexes per query, the problem complexity increases stronger than we can soften it with our decomposition approach. Therefore, a combination of our hybrid and decomposition approach is advisable.

7.4.4. Risk-Averse Mean-Variance Optimization

In this section, we evaluate our BQP-based risk-averse approach (see Section 7.3.3) for the index selection problem. We have previously shown how we were able to reduce the calculation time and, thus, calculate robust index selections for TPC-DS [194]. In this section, we want to demonstrate the effects of using mean-variance optimization (MVO). To better understand the BQP solver's index selection, we use the less complex TPC-H benchmark and allow only indexes of width 1 and a single index per query. Further, for simplicity, we use only model results and neglect possible improvements of actual results (by applying an increasing number of indexes per query that are not covered by the model input). Figure 7.3 visualizes the cost savings and index sizes of all significant indexes that save more than 0.1% of the overall workload costs.

There are six indexes (i.e., index 1,..., index 6), which have a (significant) effect on nine queries and whose sizes range from 0.21 to 1.57 GB. Indexes have an effect on different queries. In the following, we assume an index budget of 4 GB. We want to optimize the index selection for $S = 10$ different workload scenarios. The probability P_s of all scenarios and the expected frequency $f_{j,s}$ of all queries is the same, i.e., $P_s = 1/10, s = 1, ..., 10$ and $\sum_{s=1,...,S} f_{j,s} = 10,\ j = 1, ..., Q$. Recap,

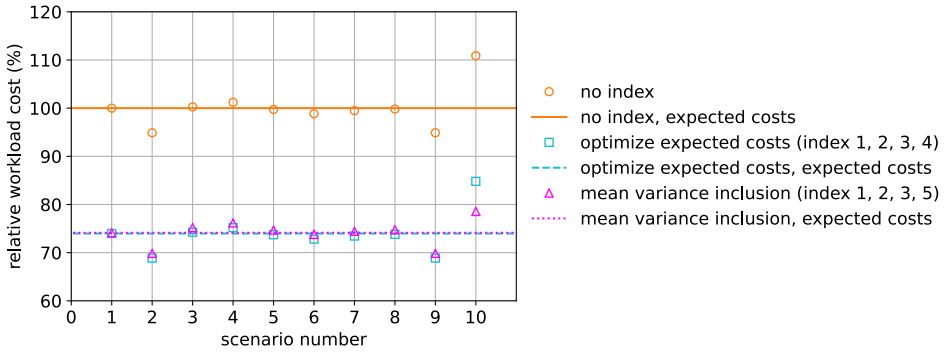


Figure 7.4.: Index selection results for the basic (i.e., optimizing expected costs) and risk-averse (including mean-variance) ILP-based approach for TPC-H (SF 10) and a budget of 4 GB; max index width: 1, max indexed per query: 1.

when optimizing the expected workload costs, the individual query frequencies per scenario $f_{j,s}$ do not matter.

For this small problem instance, we can manually determine the optimal solution that optimizes the expected costs: Index 1 ($3.5 + 7 + 7.5\%$ savings) and index 2 ($2.1 + 2.1\%$ savings) dominate the cost savings and leave $0.86 = 4 - (1.57 + 1.57)$ of the assumed 4 GB budget; For the remaining budget, we can select any two of the indexes 3 - 5; Choosing index 3 (1% savings) and index 4 (0.9% savings) is best because index 5 saves only 0.8% for query 9 (for query 21, we use the better index 1). Note, selecting index 5 instead of index 4 (or 3) would only slightly worsen the overall cost savings, but could be a viable option, e.g., when we want to optimize the performance of a potential workload peak caused by query 9.

In the following, we assume a skewed workload frequency over the 10 scenarios for query 9, which would benefit from selecting index 5. We assume $f_{j,s} = 1$ for scenario $s = 1$, $j = 1, ..., Q$. Query 9 does not occur in the scenarios 2 - 9, i.e., $f_{j,s} = 0, j = 9, s = 2, ..., 9$. Query 9 dominates in scenario 10, i.e., $f_{j,s} = 9, j = 9, s = 10$. Further, scenario 10 has the highest workload costs without indexes. Additionally, we only vary the query frequencies of queries that are not (significantly) affected by indexes (i.e., query 1, 3, 5, 6, 7, 10). The scenario probabilities P_s and expected frequencies $f_{j,s}$ for each query are still the same.

We investigate the influence of MVO for this index selection problem. Figure 7.4 shows the expected workload costs and costs for each of the ten scenarios when (i) using no indexes, (ii) optimizing the expected costs, and (iii) considering the mean variance. The solutions for optimizing the expected and including mean variance differ in the selection of index 4 (which saves overall more costs for query 11) and index 5 (which reduces the costs of the peak workload scenario 10 caused by query 9).

Compared to the workload costs without indexes, both ILP-based approaches reduce the (expected and individual scenarios') workload costs. Optimizing the expected costs is slightly better (0.9% larger cost savings) than the MVO inclusion for scenario 2 - 9, in which query 9 does not occur, but index 4 can speed up query 11.

The savings for scenario 1 and the expected costs are 0.1%, which we regard as practically insignificantly. In contrast, the mean-variance inclusion softens the peak scenario 10 and, thus, provides an overall more robust performance.

Summary: *Using mathematical programming enables us to extend the optimization function to consider the performance variance of individual workload scenarios for the index selection problem. As a result, the performance of all scenarios is similar, and we can weaken cost peaks of individual workload scenarios. Considering the variance increases the complexity of the program because the optimization function is quadratic instead of linear. However, the BQP is compatible with our heuristic approaches: This way, we can keep the number of input combinations for individual programs low to reduce the overall calculation time effectively.*

7.5. Summary

Selecting secondary indexes is essential to speed up complex database workloads. Among various selection approaches, we can use mathematical programming to find suitable indexes. Mathematical programming guarantees optimal solutions for the modeled problem. Thus, it is naturally well-suited for small problem instances, e.g., out-performing state-of-the-art greedy approaches for TPC-H workloads.

For larger problems, the input generation and solver time become increasingly time-intensive and eventually practically unfeasible. Instead of naively retrieving query costs for each index combination (as pure ILP approaches), we can use our hybrid index selection approach, which uses the conducted cost estimates of existing heuristic selection approaches as input for ILP. Further, using a decomposition approach, we can decrease the solver time by first "prefilter" overall good index combinations from smaller candidate subsets. The final indexes are then selected from the best combinations. Using our ILP heuristics, we could also find better indexes than state-of-the-art approaches for larger TPC-DS workloads, for which the best selections contain wide indexes and require multiple indexes per query. Further, by reducing the calculation time, we can tackle not only larger problem instances but also more complex formulations, e.g., considering the variance of multiple potential future workloads.

8

Conclusion

There is much to do, and I am busy, very busy.

Wilhelm Röntgen

In this chapter, we conclude this thesis with a summary of our contributions while reflecting on our thesis statement:

Allocation approaches based on integer linear programming can quickly calculate allocations for partially replicated database clusters with a better mix of robust performance and memory efficiency than state-of-the-art (greedy) approaches.

Further, we outline opportunities for future work.

8.1. Summary

In this thesis, we developed ILP-based allocation approaches for partially replicated database clusters. We introduced programming models for the basic read-only problem and extensions to cope with node failures, workload uncertainty, data modifications, and reallocations. Because ILP is not scaleable, we proposed three heuristics, which can be flexibly combined to balance the solution quality and calculation time: (i) decomposing a single complex ILP into multiple easier-to-solve subprograms, (ii) clustering queries with minor workload shares, and (iii) using solver's optimality gap or time limit. We evaluated our allocation models for three workloads, which comprise thousands of query classes and hundreds of data fragments, and compared them to state-of-the-art approaches, mainly the greedy approach proposed by Rabl and Jacobsen [181]. While the results naturally depend on the specific input, we could calculate allocations with a significantly lower memory consumption, requiring up to 46% less data compared to the greedy state-of-the-art approach while equally balancing the workload. Further, our allocations reduce the maximum workload share of nodes in case of failures or uncertain workloads, which results in a higher query throughput as our end-to-end evaluations show.

We also investigated ILP approaches for selecting database indexes, particularly based on the optimizer's cost estimates. Optimal solutions are only tractable for small problem instances. In other cases, we must limit the index width or the

maximum number of applicable indexes per query. If the calculation time becomes too high, we can (i) use existing index selection approaches for the ILP's input generation and (ii) decompose the problem by first preselecting index combinations out of smaller input subsets and finally computing the result based on selected combinations per subset. Using our hybrid and decomposition approach, we can also solve more complex index selection formulations with wider indexes, more indexes per query, or even consider the performance variance of multiple workload scenarios. As a result, we could calculate comparable and even better results than the greedy *Extend* algorithm [196], which has shown good results for various inputs [123].

8.2. Future Work

We covered many aspects of ILP-based heuristics for partially replicated database clusters and selecting indexes. However, we still see several possibilities for future work in this research area:

Combination of extensions and automatic tuning. In this thesis, we addressed allocation extensions for partially replicated database clusters individually. For ILP-based heuristics, future work could investigate how to combine them while balancing solution quality and calculation time. Likewise, automatically tuning ILP-based heuristics to lower the computation time is challenging (i.e., deciding on a good combination of decomposition, chunking, and solver relaxations). Recap, it is difficult (or impossible) to predict an ILP's runtime (without setting the solver's time limit). Hence, finding a suitable chunk decomposition and clustering for a bounded computation time is also difficult. Future work could investigate how to handle situations with limited calculation times, e.g., starting with a quick solution first.

System integration. Besides aspects regarding the allocation's calculation, there are further challenges when using partially replicated database clusters (see Section 6.5): We have to cope with inaccuracies of the model input, also due to changing workloads and underlying data. We think that robustness extensions (e.g., considering node failures and uncertain workloads) are essential for allocations in practice. In this context, further end-to-end evaluations (also for different database systems) can help to find and quantify potential model inaccuracies, which may also demand additional model extensions. Depending on the database systems used, model adaptions may be required (e.g., to better match the system's reallocation costs or to incorporate distributed query processing). Finally, in practice, we have to monitor the system for input changes, reevaluate current allocations, and, if necessary, reallocate data.

Index candidate selection. We see optimization possibilities for our hybrid index selection approach. We could evaluate multiple index selection algorithms for a

diverse candidate generation and investigate which specific heuristics or adaptions to use to generate the ILP model's input. Our index selection evaluation platform [122] facilitates such studies.

Index selection in practice. In this thesis, we evaluated index selection algorithms for static read-only analytical workloads based on cost estimates. Bringing our findings to practice poses some challenges for future work. First, we must handle inaccurate cost estimates, e.g., by using learned index benefits [203] or adjusting selections based on actual measurements. Further, future work can investigate transactional workloads, incorporating index maintenance costs. Finally, we must deal with index reconfigurations for changing workloads [192, 193], performance robustness of the workload [184], and joint tuning with other physical design aspects [44, 123].

Overall, database system optimization remains a large and exciting research area, and we are happy to contribute to this field.

Appendix A

List of Publications

Our thesis' main contributions have been published at the international conferences CIKM, EDBT, ICDE, SSDBM, and VLDB. We further contributed to the field of columnar in-memory database systems [66], replication systems and middleware [107, 119, 199], data-driven what-if analyses [35, 118] and available-to-promise checks [216], and cached/materialized aggregates [159, 160, 161, 162]. In the following, we list all our publications ordered by date from newest to oldest and highlight the publications that are closely related to this thesis.

- **Stefan Halfpap and Rainer Schlosser. Fragment allocations for partially replicated databases considering data modifications and changing workloads.** Under submission, 12 pages.
- **Stefan Halfpap. Hybrid index selection using integer linear programming based on cached cost estimates of heuristic approaches.** In Proceedings of the International Workshop on Simplicity in Management of Data (SiMoD), pages 5:1–5:4, 2023.
- Peter Boncz, Yannis Chronis, Jan Finis, Stefan Halfpap, Viktor Leis, Thomas Neumann, Anisoara Nica, Caetano Sauer, Knut Stolze, Marcin Zukowski. SPA: Economical and workload-driven indexing for data analytics in the cloud. In Proceedings of the International Conference on Data Engineering (ICDE), pages 3740–3746, 2023.
- **Stefan Halfpap and Rainer Schlosser. Memory-efficient database fragment allocation for robust load balancing when nodes fail.** In Proceedings of the International Conference on Data Engineering (ICDE), pages 1811–1816, 2021.
- **Rainer Schlosser and Stefan Halfpap. Robust and memory-efficient database fragment allocation for large and uncertain database workloads.** In Proceedings of the International Conference on Extending Database Technology (EDBT), pages 367–372, 2021.
- **Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. Magic mirror in my hand, which is the best in the land? An**

experimental evaluation of index selection algorithms. Proceedings of the VLDB Endowment, 13(11): 2382–2395, 2020.

- **Stefan Halfpap and Rainer Schlosser. Exploration of dynamic query-based load balancing for partially replicated database systems with node failures.** In Proceedings of the International Conference on Information and Knowledge Management (CIKM), pages 3409–3412, 2020.
- **Rainer Schlosser and Stefan Halfpap. A decomposition approach for risk-averse index selection.** In Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM), pages 16:1–16:4, 2020.
- **Stefan Halfpap. Efficient scale-out using query-driven workload distribution and fragment allocation.** In Proceedings of the VLDB PhD Workshop, 2019.
- **Stefan Halfpap and Rainer Schlosser. Workload-driven fragment allocation for partially replicated databases using linear programming.** In Proceedings of the International Conference on Data Engineering (ICDE), pages 1746–1749, 2019.
- **Stefan Halfpap and Rainer Schlosser. A comparison of allocation algorithms for partially replicated databases.** In Proceedings of the International Conference on Data Engineering (ICDE), pages 2008–2011, 2019.
- Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. Hyrise re-engineered: An extensible database system for research in relational in-memory data management. In Proceedings of the International Conference on Extending Database Technology (EDBT), pages 313–324, 2019.
- Stefan Klauck, Max Plauth, Sven Knebel, Marius Strobl, Douglas Santry, and Lars Eggert. Eliminating the bandwidth bottleneck of central query dispatching through TCP connection hand-over. In Proceedings of the Conference Datenbanksysteme für Business, Technologie und Web (BTW), pages 97–106, 2019.
- Jens Hiller, Maël Kimmerlin, Max Plauth, Seppo Heikkilä, Stefan Klauck, Ville Lindfors, Felix Eberhardt, Dariusz Bursztynowski, Jesus Llorente Santos, Oliver Hohlfeld, and Klaus Wehrle. Giving customers control over their data: Integrating a policy language into the cloud. In Proceedings of the International Conference on Cloud Engineering (IC2E), pages 241–249, 2018.
- David Schwalb, Jan Kossmann, Martin Faust, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. Hyrise-R: Scale-out and hot-standby through lazy master replication for enterprise applications. In Proceedings of the International

Workshop on In-Memory Data Management and Analytics (IMDM), pages 7:1–7:7, 2015.

- Stefan Klauck, Lars Butzmann, Stephan Müller, Martin Faust, David Schwalb, Matthias Uflacker, Werner Sinzig, and Hasso Plattner. Interactive, flexible, and generic what-if analyses using in-memory column stores. In Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA), pages 488–497, 2015.

- Stephan Müller, Anisoara Nica, Lars Butzmann, Stefan Klauck, and Hasso Plattner. Using object-awareness to optimize join processing in the SAP HANA aggregate cache. In Proceedings of the International Conference on Extending Database Technology (EDBT), pages 557–568, 2015.

- Lars Butzmann, Stefan Klauck, Stephan Müller, Matthias Uflacker, Werner Sinzig, and Hasso Plattner. Generic business simulation using an in-memory column store. In Proceedings of the Conference Datenbanksysteme für Business, Technologie und Web (BTW), pages 633–643, 2015.

- Stephan Müller, Lars Butzmann, Stefan Klauck, and Hasso Plattner. An adaptive aggregate maintenance approach for mixed workloads in columnar in-memory databases. In Proceedings of the Australasian Computer Science Conference (ACSC), pages 3–12, 2014.

- Stephan Müller, Lars Butzmann, Stefan Klauck, and Hasso Plattner. Workload-aware aggregate maintenance in columnar in-memory databases. In Proceedings of the International Conference on Big Data (BigData), pages 62–69, 2013.

- Stephan Müller, Lars Butzmann, Kai Höwelmeyer, Stefan Klauck, and Hasso Plattner. Efficient view maintenance for enterprise applications in columnar in-memory databases. In Proceedings of the International Enterprise Distributed Object Computing Conference (EDOC), pages 249–258, 2013.

- Christian Tinnefeld, Stephan Müller, Helen Kaltegärtner, Sebastian Hillig, Lars Butzmann, David Eickhoff, Stefan Klauck, Daniel Taschik, Björn Wagner, Oliver Xylander, Alexander Zeier, Hasso Plattner, and Cafer Tosun. Available-to-promise on an in-memory column store. In Proceedings of the Conference Datenbanksysteme für Business, Technologie und Web (BTW), pages 667–686, 2011.

Bibliography

[1] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database tuning advisor for Microsoft SQL Server 2005. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1110–1121, 2004.

[2] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 169–180, 2001.

[3] Maria Albareda-Sambola and Elena Fernandez. The stochastic generalized assignment problem with bernoulli demand. *TOP*, 8(2):165–190, 2000.

[4] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. Adaptive range filters for cold data: Avoiding trips to siberia. *Proceedings of the VLDB Endowment*, 6(14):1714–1725, 2013.

[5] Peter M. G. Apers. Redundant allocation of relations in a communication network. In *Proceedings of the Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 245–258, 1981.

[6] Peter M. G. Apers. Data allocation in distributed database systems. *ACM Transactions on Database Systems (TODS)*, 13(3):263–304, 1988.

[7] Aaron Archer, Kevin Aydin, MohammadHossein Bateni, Vahab S. Mirrokni, Aaron Schild, Ray Yang, and Richard Zhuang. Cache-aware load balancing of data center applications. *Proceedings of the VLDB Endowment*, 12(6):709–723, 2019.

[8] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1753–1758, 2017.

[9] Joy Arulraj, Andrew Pavlo, and Subramanya Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 707–722, 2015.

[10] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings*

of the International Conference on Management of Data (SIGMOD), pages 583–598, 2016.

[11] Kevin Aydin, MohammadHossein Bateni, and Vahab S. Mirrokni. Distributed balanced partitioning via linear embedding. In *Proceedings of the International Conference on Web Search and Data Mining (WSDM)*, pages 387–396, 2016.

[12] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. AnyDB: An architecture-less DBMS for any workload. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2021.

[13] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[14] John J. Bartholdi and Loren K. Platzman. Heuristics based on spacefilling curves for combinatorial problems in euclidean space. *Management Science*, 34(3):291–305, 1988.

[15] William J. Baumol and Philip Wolfe. A warehouse-location problem. *Operations Research*, 6(2):252–263, 1958.

[16] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. Photon: A fast query engine for lakehouse systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 2326–2339, 2022.

[17] Björn Bergsten, Michel Couprie, and Patrick Valduriez. Overview of parallel architectures for databases. *The Computer Journal*, 36(8):734–740, 1993.

[18] Michel Berkelaar. lp_solve: A mixed integer linear programming solver. URL: `https://lpsolve.sourceforge.net/`, 1997. Accessed: April 1, 2023.

[19] Daniel Bernoulli. Exposition of a new theory on the measurement of risk. In *The Kelly capital growth investment criterion: Theory and practice*, pages 11–24. 2011.

[20] Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske,

Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 8.0. ZIB-Report 21–41, 2021.

[21] Robert E. Bixby. Solving real-world linear programs: A decade and more of progress. *Operations Research*, 50(1):3–15, 2002.

[22] Robert E. Bixby. A brief history of linear and mixed-integer programming computation. *Documenta Mathematica*, 2012:107–121, 2012.

[23] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[24] Christian Blum, Jakob Puchinger, Günther R. Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11(6):4135–4151, 2011.

[25] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[26] Martin Boissier. Robust and budget-constrained encoding configurations for in-memory database systems. *Proceedings of the VLDB Endowment*, 15(4):780–793, 2021.

[27] Martin Boissier, Carsten Alexander Meyer, Timo Djürken, Jan Lindemann, Kathrin Mao, Pascal Reinhardt, Tim Specht, Tim Zimmermann, and Matthias Uflacker. Analyzing data relevance and access patterns of live production database systems. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 2473–2475, 2016.

[28] Martin Boissier, Rainer Schlosser, and Matthias Uflacker. Hybrid data layouts for tiered HTAP databases with pareto-optimal data placements. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 209–220, 2018.

[29] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple but effective techniques for NUMA memory management. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, pages 19–31, 1989.

[30] Angela Bonifati, Fabiano Cattaneo, Stefano Ceri, Alfonso Fuggetta, and Stefano Paraboschi. Designing data marts for data warehouses. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(4):452–483, 2001.

[31] Haran Boral, William Alexander, Larry Clay, George P. Copeland, Scott Danforth, Michael J. Franklin, Brian E. Hart, Marc G. Smith, and Patrick Valduriez. Prototyping Bubba, A highly parallel database system. *IEEE Transactions on Knowledge & Data Engineering*, 2(1):4–24, 1990.

[32] Renata Borovica, Ioannis Alagiannis, and Anastasia Ailamaki. Automated physical designers: What you see is (not) what you get. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*, pages 9:1–9:6, 2012.

[33] A. L. Brearley, Gautam Mitra, and H. Paul Williams. Analysis of mathematical programming problems prior to applying the simplex algorithm. *Mathematical Programming*, 8(1):54–83, 1975.

[34] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 227–238, 2005.

[35] Lars Butzmann, Stefan Klauck, Stephan Müller, Matthias Uflacker, Werner Sinzig, and Hasso Plattner. Generic business simulation using an in-memory column store. In *Proceedings of the Conference Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 633–643, 2015.

[36] Alberto Caprara, Matteo Fischetti, and Dario Maio. Exact and approximate algorithms for the index selection problem in physical database design. *IEEE Transactions on Knowledge & Data Engineering*, 7(6):955–967, 1995.

[37] Alberto Caprara and Juan José Salazar González. A branch-and-cut algorithm for a generalization of the uncapacitated facility location problem. *TOP*, 4:135–163, 1996.

[38] Richard G. Casey. Allocation of copies of a file in an information network. In *AFIPS Conference Proceedings*, pages 617–625, 1972.

[39] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 9–18, 2004.

[40] Stefano Ceri, Giancarlo Martella, and Giuseppe Pelagatti. Optimal file allocation in a computer network: a solution method based on the knapsack problem. *Computer Networks*, 6(5):345–357, 1982.

[41] Prima Chairunnanda, Khuzaima Daudjee, and M. Tamer Özsu. ConfluxDB: Multi-master replication for partitioned snapshot isolation databases. *Proceedings of the VLDB Endowment*, 7(11):947–958, 2014.

[42] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 146–155, 1997.

[43] Surajit Chaudhuri and Vivek R. Narasayya. Index merging. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 296–303, 1999.

[44] Surajit Chaudhuri and Vivek R. Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 3–14, 2007.

[45] Wesley W. Chu. Optimal file allocation in a multiple computer system. *IEEE Transactions on Computers*, 18(10):885–889, 1969.

[46] Mariano P. Consens, Kleoni Ioannidou, Jeff LeFevre, and Neoklis Polyzotis. Divergent physical design tuning for replicated databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 49–60, 2012.

[47] George P. Copeland, William Alexander, Ellen E. Boughter, and Tom W. Keller. Data placement in Bubba. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 99–108, 1988.

[48] George P. Copeland and Setrag Khoshafian. A decomposition storage model. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 268–279, 1985.

[49] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 251–264, 2012.

[50] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. MIT Press, 2009.

[51] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design, Fifth Edition*. Addison-Wesley, 2011.

[52] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. Citus: Distributed PostgreSQL for data-intensive applications. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 2490–2502, 2021.

[53] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1):48–57, 2010.

[54] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven

Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The Snowflake elastic data warehouse. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 215–226, 2016.

[55] R. J. Dakin. A tree-search algorithm for mixed integer programming problems. *The Computer Journal*, 8(3):250–255, 1965.

[56] George B. Dantzig and Mukund N. Thapa. *Linear Programming 1: Introduction*. Springer Science & Business Media, 1997.

[57] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. CoPhy: A scalable, portable, and interactive index advisor for large workloads. *Proceedings of the VLDB Endowment*, 4(6):362–372, 2011.

[58] Mohammad Dashti, Alexandra Fedorova, Justin R. Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. Traffic management: A holistic approach to memory placement on NUMA systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 381–394, 2013.

[59] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 715–726, 2006.

[60] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. GAMMA - A high performance dataflow database machine. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 228–237, 1986.

[61] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge & Data Engineering*, 2(1):44–62, 1990.

[62] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[63] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. AI meets AI: Leveraging query executions to improve index recommendations. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1241–1258, 2019.

[64] Lawrence W. Dowdy and Derrell V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, 1982.

[65] Ulrich Drepper. What every programmer should know about memory. URL: `https://people.freebsd.org/~lstewart/articles/cpumemory.pdf`, 2007. Accessed: April 1, 2023.

[66] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. Hyrise re-engineered: An extensible database system for research in relational in-memory data management. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 313–324, 2019.

[67] Cees Duin and Stefan Voß. The pilot method: A strategy for heuristic repetition with application to the steiner problem in graphs. *Networks*, 34(3):181–191, 1999.

[68] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, and Tim Kraska. Revisiting reuse in main memory database systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1275–1289, 2017.

[69] Anshuman Dutt and Jayant R. Haritsa. Plan bouquets: Query processing without selectivity estimation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1039–1050, 2014.

[70] M. A. Efroymson and T. L. Ray. A branch-bound algorithm for plant location. *Operations Research*, 14(3):361–368, 1966.

[71] Sameh Elnikety, Steven G. Dropsho, and Willy Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In *Proceedings of the EuroSys Conference*, pages 399–412, 2007.

[72] Cristian Estan and Jeffrey F. Naughton. End-biased samples for join cardinality estimation. In *Proceedings of the International Conference on Data Engineering (ICDE)*, page 20, 2006.

[73] Kapali P. Eswaran. Placement of records in a file and file allocation in a computer. In *Proceedings of the IFIP Congress*, pages 304–307, 1974.

[74] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA database – an architecture overview. *IEEE Data Engineering Bulletin*, 35(1):28–33, 2012.

[75] Martin Faust, Martin Boissier, Marvin Keller, David Schwalb, Holger Bischoff, Katrin Eisenreich, Franz Färber, and Hasso Plattner. Footprint reduction and uniqueness enforcement with hash indices in SAP HANA. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, volume 9828, pages 137–151, 2016.

[76] E. Feldman, F. A. Lehrer, and T. L. Ray. Warehouse location under continuous economies of scale. *Management Science*, 12(9):670–684, 1966.

[77] FICO. FICO Xpress. URL: `https://www.fico.com/en/products/fico-xpress-optimization`, 1983. Accessed: April 1, 2023.

[78] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Thomson/Brooks/Cole, 2003.

[79] Florian Funke, Alfons Kemper, and Thomas Neumann. Compacting transactional data in hybrid OLTP & OLAP databases. *Proceedings of the VLDB Endowment*, 5(11):1424–1435, 2012.

[80] Mainak Ghosh, Ashwini Raina, Le Xu, Xiaoyao Qian, Indranil Gupta, and Himanshu Gupta. Popular is cheaper: Curtailing memory costs in interactive analytics engines. In *Proceedings of the EuroSys Conference*, pages 40:1–40:14, 2018.

[81] Fred W. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986.

[82] Fred W. Glover. Tabu search - part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[83] Fred W. Glover. Tabu search - part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[84] Fred W. Glover and Manuel Laguna. *Tabu Search*. Kluwer, 1997.

[85] Fred W. Glover and Kenneth Sörensen. Metaheuristics. *Scholarpedia*, 10(4):6532, 2015.

[86] GNU Project. GNU Linear Programming Kit (GLPK). URL: `https://www.gnu.org/software/glpk/`, 2000. Accessed: April 1, 2023.

[87] James R. Goodman and Philip J. Woest. The Wisconsin multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 422–431, 1988.

[88] Goetz Graefe. B-tree indexes for high update rates. *SIGMOD Record*, 35(1):39–44, 2006.

[89] Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics*, 25:1–16, 2020.

[90] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis E. Shasha. The dangers of replication and a solution. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 173–182, 1996.

[91] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A main memory hybrid storage engine. *Proceedings of the VLDB Endowment*, 4(2):105–116, 2010.

[92] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon Redshift and the case for simpler data warehouses. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1917–1923, 2015.

[93] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.

[94] Gurobi Optimization. Gurobi optimizer (Gurobi). URL: `https://www.gurobi.com`, 2009. Accessed: April 1, 2023.

[95] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Arun N. Swami. Selectivity and cost estimation for joins based on random sampling. *Journal of Computer and System Sciences*, 52(3):550–569, 1996.

[96] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E. Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM allocation service at scale. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 845–861, 2020.

[97] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[98] Stefan Halfpap. Efficient scale-out using query-driven workload distribution and fragment allocation. In *Proceedings of the VLDB PhD Workshop*, 2019.

[99] Stefan Halfpap. Implementation and model input parameters for partially replicated database clusters. URL: `https://hyrise.github.io/replication/`, 2019. Accessed: April 1, 2023.

[100] Stefan Halfpap. Hybrid index selection using integer linear programming based on cached cost estimates of heuristic approaches. In *Proceedings of the International Workshop on Simplicity in Management of Data (SiMoD)*, pages 5:1–5:4, 2023.

[101] Stefan Halfpap and Rainer Schlosser. A comparison of allocation algorithms for partially replicated databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 2008–2011, 2019.

[102] Stefan Halfpap and Rainer Schlosser. Workload-driven fragment allocation for partially replicated databases using linear programming. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1746–1749, 2019.

[103] Stefan Halfpap and Rainer Schlosser. Exploration of dynamic query-based load balancing for partially replicated database systems with node failures. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 3409–3412, 2020.

[104] Stefan Halfpap and Rainer Schlosser. Memory-efficient database fragment allocation for robust load balancing when nodes fail. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1811–1816, 2021.

[105] Brad Hammond. Merge replication in Microsoft's SQL Server 7.0. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, page 527, 1999.

[106] Willem K. Klein Haneveld and Maarten H. van der Vlerk. Stochastic integer programming: General models and algorithms. *Annals of Operations Research*, 85:39–57, 1999.

[107] Jens Hiller, Maël Kimmerlin, Max Plauth, Seppo Heikkilä, Stefan Klauck, Ville Lindfors, Felix Eberhardt, Dariusz Bursztynowski, Jesus Llorente Santos, Oliver Hohlfeld, and Klaus Wehrle. Giving customers control over their data: Integrating a policy language into the cloud. In *Proceedings of the International Conference on Cloud Engineering (IC2E)*, pages 241–249, 2018.

[108] Hui-I Hsiao and David J. DeWitt. Chained declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 456–465, 1990.

[109] IBM. IBM ILOG CPLEX Optimization Studio. URL: `https://www.ibm.com/products/ilog-cplex-optimization-studio`, 1988. Accessed: April 1, 2023.

[110] Yannis E. Ioannidis. The history of histograms (abridged). In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 19–30, 2003.

[111] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[112] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 134–143, 2000.

[113] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 195–206, 2011.

[114] Tim Kiefer, Benjamin Schlegel, and Wolfgang Lehner. Experimental evaluation of NUMA effects on database management systems. In *Proceedings of the Conference Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 185–204, 2013.

[115] Jeremie S. Kim, Minesh Patel, Abdullah Giray Yaglikçi, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting rowhammer: An experimental analysis of modern DRAM devices and mitigation techniques. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 638–651, 2020.

[116] Scott Kirkpatrick, C. Daniel Gelatt Jr., and Mario P. Vecchi. Optimization by simmulated annealing. *Science*, 220(4598):671–680, 1983.

[117] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. ERIS: A numa-aware in-memory storage engine for analytical workload. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 74–85, 2014.

[118] Stefan Klauck, Lars Butzmann, Stephan Müller, Martin Faust, David Schwalb, Matthias Uflacker, Werner Sinzig, and Hasso Plattner. Interactive, flexible, and generic what-if analyses using in-memory column stores. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 488–497, 2015.

[119] Stefan Klauck, Max Plauth, Sven Knebel, Marius Strobl, Douglas Santry, and Lars Eggert. Eliminating the bandwidth bottleneck of central query dispatching through TCP connection hand-over. In *Proceedings of the Conference Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 97–106, 2019.

[120] Konstantin Kogan and Avraham Shtub. DGAP - the dynamic generalized assignment problem. *Annals of Operations Research*, 69:227–239, 1997.

[121] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. Farview: Disaggregated memory with

operator off-loading for database engines. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2022.

[122] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. Index selection evaluation platform. URL: `https://git.io/index_selection_evaluation`, 2020. Accessed: April 1, 2023.

[123] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. Magic mirror in my hand, which is the best in the land? An experimental evaluation of index selection algorithms. *Proceedings of the VLDB Endowment*, 13(11):2382–2395, 2020.

[124] Jan Kossmann, Alexander Kastius, and Rainer Schlosser. SWIRL: Selection of workload-aware indexes using reinforcement learning. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 2:155–2:168, 2022.

[125] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast updates on read-optimized databases using multi-core CPUs. *Proceedings of the VLDB Endowment*, 5(1):61–72, 2011.

[126] Alfred A. Kuehn and Michael J. Hamburger. A heuristic program for locating warehouses. *Management Science*, 9:643–666, 1963.

[127] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[128] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The Vertica analytic database: C-Store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[129] Hai Lan, Zhifeng Bao, and Yuwei Peng. An index advisor using deep reinforcement learning. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 2105–2108, 2020.

[130] Ailsa H. Land and Alison G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[131] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. Mtcache: Transparent mid-tier database caching in SQL server. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 177–188, 2004.

[132] Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. Cost modelling for optimal data placement in heterogeneous main memory. *Proceedings of the VLDB Endowment*, 15(11):2867–2880, 2022.

[133] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, Wook-Shin Han, Chang Gyoo Park, Hyoung Jun Na, and Joo-Yeon Lee. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. *Proceedings of the VLDB Endowment*, 10(12):1598–1609, 2017.

[134] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 743–754, 2014.

[135] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[136] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. Leanstore: In-memory data management beyond main memory. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 185–196, 2018.

[137] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John L. Hennessy, Mark Horowitz, and Monica S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[138] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. Identifying hot and cold data in main-memory databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 26–37, 2013.

[139] K. Dan Levin and Howard Lee Morgan. Optimizing distributed data bases: a framework for research. In *AFIPS Conference Proceedings*, volume 44, pages 473–478, 1975.

[140] Jiexing Li, Jeffrey F. Naughton, and Rimma V. Nehme. Resource bricolage for parallel database systems. *Proceedings of the VLDB Endowment*, 8(1):25–36, 2014.

[141] Gabriel Paludo Licks, Júlia Mara Colleoni Couto, Priscilla de Fátima Miehe, Renata De Paris, Duncan Dubugras A. Ruiz, and Felipe Meneguzzi. SmartIX: A database indexing agent based on reinforcement learning. *Applied Intelligence*, 50(8):2575–2588, 2020.

[142] Shen Lin and Brian W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, 1973.

[143] Miron Livny, Setrag Khoshafian, and Haran Boral. Multi-disk management algorithms. *IEEE Data Engineering Bulletin*, 9(1):24–36, 1986.

[144] David B. Lomet. Cost/performance in modern data stores: How data caching systems succeed. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 9:1–9:10, 2018.

[145] Vincent Y. Lum and Huei Ling. An optimization problem on the selection of secondary keys. In *Proceedings of the ACM Annual Conference*, pages 349–356, 1971.

[146] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier database caching for e-business. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 600–611, 2002.

[147] Samy A. Mahmoud and J. Spruce Riordon. Optimal allocation of resources in distributed information networks. *ACM Transactions on Database Systems (TODS)*, 1(1):66–78, 1976.

[148] Jack A. Mandelman, Robert H. Dennard, Gary B. Bronner, John K. DeBrosse, Rama Divakaruni, Yujun Li, and Carl J. Raden. Challenges and future directions for the scaling of dynamic random-access memory (DRAM). *IBM Journal of Research and Development*, 46(2-3):187–222, 2002.

[149] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB Journal*, 9(3):231–246, 2000.

[150] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 191–202, 2002.

[151] Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. NashDB: An end-to-end economic method for elastic database fragmentation, replication, and provisioning. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1253–1267, 2018.

[152] Ryan C. Marcus and Olga Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *Proceedings of the VLDB Endowment*, 12(11):1733–1746, 2019.

[153] Harry Markowitz. Portfolio selection. *The Journal of Finance*, 7(1):77–91, 1952.

[154] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics, Second Edition*. Springer, 2004.

[155] Jesús M. Milán-Franco, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Bettina Kemme. Adaptive middleware for data replication. In *Proceedings of the International Middleware Conference (Middleware)*, pages 175–194, 2004.

[156] Guido Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 476–487, 1998.

[157] Howard L. Morgan and K. Dan Levin. Optimal program and data locations in computer networks. *Communications of the ACM*, 20(5):315–322, 1977.

[158] Tobias Mühlbauer, Wolf Rödiger, Angelika Reiser, Alfons Kemper, and Thomas Neumann. ScyPer: Elastic OLAP throughput on transactional data. In *Proceedings of the International Workshop on Data analytics in the Cloud (DanaC)*, pages 11–15, 2013.

[159] Stephan Müller, Lars Butzmann, Kai Höwelmeyer, Stefan Klauck, and Hasso Plattner. Efficient view maintenance for enterprise applications in columnar in-memory databases. In *Proceedings of the International Enterprise Distributed Object Computing Conference (EDOC)*, pages 249–258, 2013.

[160] Stephan Müller, Lars Butzmann, Stefan Klauck, and Hasso Plattner. Workload-aware aggregate maintenance in columnar in-memory databases. In *Proceedings of the International Conference on Big Data (BigData)*, pages 62–69, 2013.

[161] Stephan Müller, Lars Butzmann, Stefan Klauck, and Hasso Plattner. An adaptive aggregate maintenance approach for mixed workloads in columnar in-memory databases. In *Proceedings of the Australasian Computer Science Conference (ACSC)*, pages 3–12, 2014.

[162] Stephan Müller, Anisoara Nica, Lars Butzmann, Stefan Klauck, and Hasso Plattner. Using object-awareness to optimize join processing in the SAP HANA aggregate cache. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 557–568, 2015.

[163] Thomas Neumann and Michael J. Freitag. Umbra: A disk-based system with in-memory performance. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2020.

[164] Thomas Neumann and César A. Galindo-Legaria. Taking the edge off cardinality estimation errors using incremental execution. In *Proceedings of the Conference Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 73–92, 2013.

[165] Temel Öncan. A survey of the generalized assignment problem and its applications. *Information Systems and Operational Research (INFOR)*, 45(3):123–141, 2007.

[166] Patrick E. O'Neil and Dallan Quass. Improved query performance with variant indexes. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 38–49, 1997.

[167] Ibrahim H. Osman and Gilbert Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 63(5):511–623, 1996.

[168] Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. Workload-aware storage layout for database systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 939–950, 2010.

[169] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[170] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Fourth Edition*. Springer, 2020.

[171] Frank P. Palermo. A quantitative approach to the selection of secondary indexes. In *IBM Research RJ 730*, 1970.

[172] Stratos Papadomanolakis and Anastassia Ailamaki. An integer linear programming approach to database design. In *Proceedings of the International Conference on Data Engineering (ICDE) Workshops*, pages 442–449, 2007.

[173] Stratos Papadomanolakis, Debabrata Dash, and Anastassia Ailamaki. Efficient use of the query optimizer for automated database design. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1093–1104, 2007.

[174] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[175] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the International Middleware Conference (Middleware)*, pages 155–174, 2004.

[176] Christian Plattner, Gustavo Alonso, and M. Tamer Özsu. Extending dbmss with satellite databases. *VLDB Journal*, 17(4):657–682, 2008.

[177] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1–2, 2009.

[178] Hasso Plattner. The impact of columnar in-memory databases on enterprise systems. *Proceedings of the VLDB Endowment*, 7(13):1722–1729, 2014.

[179] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. SWORD: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 430–441, 2013.

[180] Tilmann Rabl. *Efficiency in Cluster Database Systems - Dynamic and Workload-Aware Scaling and Allocation*. PhD thesis, University of Passau, Germany, 2011.

[181] Tilmann Rabl and Hans-Arno Jacobsen. Query centric partitioning and allocation for partially replicated database systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 315–330, 2017.

[182] C. V. Ramamoorthy and Benjamin W. Wah. The isomorphism of simple file allocation. *IEEE Transactions on Computers*, 32(3):221–232, 1983.

[183] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A case for fractured mirrors. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 430–441, 2002.

[184] Keven Richly, Rainer Schlosser, and Martin Boissier. Joint index, sorting, and compression optimization for memory-efficient spatio-temporal data management. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1901–1906, 2021.

[185] Wolf Rödiger, Tobias Mühlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 592–603, 2014.

[186] Uwe Röhm, Klemens Böhm, and Hans-Jörg Schek. OLAP query routing and physical design in a database cluster. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 254–268, 2000.

[187] Uwe Röhm, Klemens Böhm, and Hans-Jörg Schek. Cache-aware query routing in a cluster of databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 641–650, 2001.

[188] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. FAS - A freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 754–765, 2002.

[189] Julien Rouhaud. HypoPG - Hypothetical indexes for PostgreSQL. URL: `https://github.com/HypoPG/hypopg`, 2015. Accessed: April 1, 2023.

[190] Florin Rusu and Alin Dobra. Sketches for size of join estimation. *ACM Transactions on Database Systems (TODS)*, 33(3):15:1–15:46, 2008.

[191] Zahra Sadri, Le Gruenwald, and Eleazar Leal. DRLindex: Deep reinforcement learning index advisor for a cluster database. In *Proceedings of the International Database Engineering & Applications Symposium (IDEAS)*, pages 11:1–11:8, 2020.

[192] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. QUIET: Continuous query-driven index tuning. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1129–1132, 2003.

[193] Kai-Uwe Sattler, Eike Schallehn, and Ingolf Geist. Autonomous query-driven index tuning. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, pages 439–448, 2004.

[194] Rainer Schlosser and Stefan Halfpap. A decomposition approach for risk-averse index selection. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 16:1–16:4, 2020.

[195] Rainer Schlosser and Stefan Halfpap. Robust and memory-efficient database fragment allocation for large and uncertain database workloads. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 367–372, 2021.

[196] Rainer Schlosser, Jan Kossmann, and Martin Boissier. Efficient scalable multi-attribute index selection using recursive strategies. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1238–1249, 2019.

[197] Rainer Schlosser, Marcel Weisgut, Leonardo Hübscher, and Oliver Nordemann. Robust index selection for stochastic dynamic workloads. *SN Computer Science*, 4(1):59:1–59:14, 2023.

[198] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. Index interactions in physical design tuning: Modeling, analysis, and applications. *Proceedings of the VLDB Endowment*, 2(1):1234–1245, 2009.

[199] David Schwalb, Jan Kossmann, Martin Faust, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. Hyrise-R: Scale-out and hot-standby through lazy master replication for enterprise applications. In *Proceedings of the International Workshop on In-Memory Data Management and Analytics (IMDM)*, pages 7:1–7:7, 2015.

[200] Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko. *High Performance MySQL - Optimization, Backups, and Replication, Third Edition.* O'Reilly, 2012.

[201] Robert Sedgewick and Kevin Wayne. *Algorithms, Fourth Edition*. Addison-Wesley, 2011.

[202] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. The case for automatic database administration using deep reinforcement learning. *CoRR*, abs/1801.05643, 2018.

[203] Jiachen Shi, Gao Cong, and Xiaoli Li. Learned index benefits: Machine learning based index performance estimation. *Proceedings of the VLDB Endowment*, 15(13):3950–3962, 2022.

[204] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in SAP HANA database: The end of a column store myth. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 731–742, 2012.

[205] E. A. Silver. An overview of heuristic solution methods. *Journal of the Operational Research Society*, 55(9):936–956, 2004.

[206] Kenneth Sörensen, Marc Sevaux, and Fred W. Glover. A history of metaheuristics. In *Handbook of Heuristics*, pages 791–808. 2018.

[207] Flávio R. C. Sousa and Javam C. Machado. Towards elastic multi-tenant database replication with quality of service. In *Proceedings of the International Conference on Utility and Cloud Computing (UCC)*, pages 168–175, 2012.

[208] William M. Spears, Kenneth A. De Jong, Thomas Bäck, David B. Fogel, and Hugo de Garis. An overview of evolutionary computation. In *Proceedings of the European Conference on Machine Learning (ECML)*, volume 667, pages 442–459, 1993.

[209] Doug Stacey. Replication: DB2, Oracle, or Sybase? *SIGMOD Record*, 24(4):95–101, 1995.

[210] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's learning optimizer. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 19–28, 2001.

[211] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[212] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-Store: Fine-grained elastic partitioning for distributed transaction processing. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[213] El-Ghazali Talbi. A taxonomy of hybrid metaheuristics. *Journal of Heuristics*, 8(5):541–564, 2002.

[214] Shreekant S. Thakkar and Mark Sweiger. Performance of an OLTP application on symmetry multiprocessor system. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 228–238, 1990.

[215] Huangshi Tian, Yunchuan Zheng, and Wei Wang. Characterizing and synthesizing task dependencies of data-parallel jobs in alibaba cloud. In *Proceedings of the Symposium on Cloud Computing (SoCC)*, pages 139–151, 2019.

[216] Christian Tinnefeld, Stephan Müller, Helen Kaltegärtner, Sebastian Hillig, Lars Butzmann, David Eickhoff, Stefan Klauck, Daniel Taschik, Björn Wagner, Oliver Xylander, Alexander Zeier, Hasso Plattner, and Cafer Tosun. Available-to-promise on an in-memory column store. In *Proceedings of the Conference Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 667–686, 2011.

[217] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In *Proceedings of the EuroSys Conference*, pages 30:1–30:14, 2020.

[218] Charles A. Trauth and Robert E. D. Woolsey. Practical aspects of integer linear programming. Technical Report SC-R-66-925, Sandia National Laboratories (SNL), 1966.

[219] Patrick Valduriez. Shared-disk architecture. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[220] Patrick Valduriez. Shared-memory architecture. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[221] Patrick Valduriez. Shared-nothing architecture. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[222] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 101–110, 2000.

[223] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1041–1052, 2017.

[224] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the EuroSys Conference*, pages 18:1–18:17, 2015.

[225] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Viktor Leis, Thomas Neumann, and Alfons Kemper. Mosaic: A budget-conscious storage engine for relational database systems. *Proceedings of the VLDB Endowment*, 13(11):2662–2675, 2020.

[226] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, pages 449–462, 2020.

[227] Florian M. Waas. Beyond conventional data warehousing - massively parallel data processing with greenplum database - (invited talk). In *Proceedings of the International Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE)*, volume 27, pages 89–96, 2008.

[228] Marcel Weisgut, Leonardo Hübscher, Oliver Nordemann, and Rainer Schlosser. Solver-based approaches for robust multi-index selection problems with stochastic workloads and reconfiguration costs. In *Proceedings of the International Conference on Operations Research and Enterprise Systems (ICORES)*, pages 28–39, 2022.

[229] Kyu-Young Whang. Index selection in relational databases. In *Proceedings of the International Conference on Foundations of Data Organization (FODO)*, pages 487–500, 1985.

[230] Eugene Wong and Randy H. Katz. Distributing a database for parallelism. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 23–29, 1983.

[231] Lik Wong, Nimar S. Arora, Lei Gao, Thuvan Hoang, and Jingwei Wu. Oracle streams: A high performance implementation for near real time asynchronous replication. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1363–1374, 2009.

[232] Robert E. D. Woolsey and Huntington S. Swanson. *Operations Research for Immediate Application: A Quick & Dirty Manual*. Harper & Row, 1975.

[233] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1081–1092, 2013.

[234] Johannes Wust, Joos-Hendrik Boese, Frank Renkes, Sebastian Blessing, Jens Krüger, and Hasso Plattner. Efficient logging for enterprise workloads on column-oriented in-memory databases. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 2085–2089, 2012.

[235] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. CS2: A new database synopsis for query estimation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 469–480, 2013.

[236] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. Locality-aware partitioning in parallel database systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 17–30, 2015.

[237] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1567–1581, 2016.

[238] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proceedings of the VLDB Endowment*, 14(10):1900–1912, 2021.

[239] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 design advisor: Integrated automatic physical database design. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1087–1097, 2004.

[240] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. Vectorwise: A vectorized analytical DBMS. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1349–1350, 2012.

Eigenständigkeitserklärung (Declaration of Authorship)

Hiermit versichere ich an Eides statt, dass die vorliegende Arbeit bisher an keiner anderen Hochschule eingereicht worden ist sowie selbständig und ausschließlich mit den angegebenen Mitteln angefertigt worden ist. Die Stellen der Arbeit, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, sind durch Angaben und Quellen kenntlich gemacht.

Potsdam, 27. April 2023

Stefan Halfpap