Efficient Scale-Out Using Query-Driven Workload Distribution and Fragment Allocation

Stefan Halfpap
Supervised by Prof. Hasso Plattner
Hasso Plattner Institute, University of Potsdam, Germany
stefan.halfpap@hpi.de

ABSTRACT

Database replication is an approach for scaling throughput and ensuring high availability. Using workload knowledge, we are able to load-balance queries to replica nodes according to the data being accessed. However, balancing the load evenly while maximizing data reuse is a challenging allocation problem. To address large-size problems, we developed a novel decomposition-based heuristic using linear programming. We compare our approach with a rule-based state-of-the-art allocation algorithm using a real-world workload comprising thousands of queries. Further, we outline how we plan to extend our approach for versatile allocation problems, e.g., considering changing workloads and robustness against node failures.

1. INTRODUCTION

Partitioning and replication are means to allow databases to process increasing workloads. Analyses of real workloads show that read-only queries account for the largest workload share [1, 11]. Scaling read-only queries is relatively simple, as we can execute them on read-only replicas without violating transactional consistency [6, 16].

Using a naive load-balancing approach, one can distribute queries independently of the accessed data. As a result, one has to store all data and apply all data modifications on all nodes. Further, queries are unlikely to profit from caching effects, because similar queries are not guaranteed to be assigned to the same replica.

In our research, we investigate query-driven workload distributions, i.e., load-balancing queries based on their accessed data. In particular, we reduce the amount of required memory on all nodes, while balancing the load evenly. Query-driven workload distributions are also beneficial to maximize caching effects. Further, when adding new nodes to a database cluster, we are able to decide which data to load first to quickly process a large share of the workload.

In practice, there are varying constraints and goals to distribute the workload and/or required data on nodes. By using linear programming (LP) we are able to address versatile allocation problems, which include robustness against potential node failures or efficient reallocations to react to changing workloads.

Proceedings of the VLDB 2019 PhD Workshop, August 26th, 2019. Los Angeles, California.

The remainder of this paper is structured as follows: We introduce the basic allocation problem in Section 2. In Section 3, we discuss related work. In Section 4, we summarize our scalable decomposition-based approach [9] to calculate workload distributions for large problem sizes. Further, we demonstrate it using a real workload. Our current research and plans for future work are described in Section 5. Section 6 concludes the paper.

2. PROBLEM DESCRIPTION

The allocation problem being examined is a coupled workload distribution and data placement problem. We assume a horizontally and/or vertically partitioned database with N disjoint data fragments. Each fragment i has a size a_i, $i = 1, ..., N$. Note, we assume a separation of the data partitioning and the allocation process, which is common [13]. In this way, existing workload-aware data partitioning algorithms can be used to generate the input fragments for our allocation approach.

Further, we assume a workload consisting of Q queries (query classes). Each query class j is defined by the subset of fragments $q_j \subseteq \{1, ..., N\}$, $j = 1, ..., Q$, it accesses. Queries account for workload shares, defined by query frequencies f_j and query costs c_j, $j = 1, ..., Q$.

Last, we assume a number of nodes K to load-balance the workload evenly. In practice, the number of nodes K can be chosen manually by a database administrator or automatically by a replication framework with regard to the desired query throughput.

We want to decide which query should be executed to which extent on a node in order to minimize the overall memory consumption for all nodes. Processing query j on node k requires to store all fragments q_j on the node. In [9], we derive an LP model to calculate optimal solutions.

Figure 1 shows an example workload and an optimal solution, i.e., an even workload distribution for $Q = 5$ queries, accessing $N = 10$ fragments, that minimizes the overall memory consumption for a database cluster with $K = 4$ nodes. (We assume an equal size for each fragment, i.e., $a_i = 1$, $i = 1, ..., N$.)

By W/V, we denote the *replication factor* of an allocation, where the total amount of data used W is normalized by the minimal amount of used data V. Because each of the ten fragments is accessed by at least one query, the minimal amount of used data is equal to the number of fragments $V = N = 10$. The replication factor of our exemplary allocation for four nodes is $W/V = 1.4$.

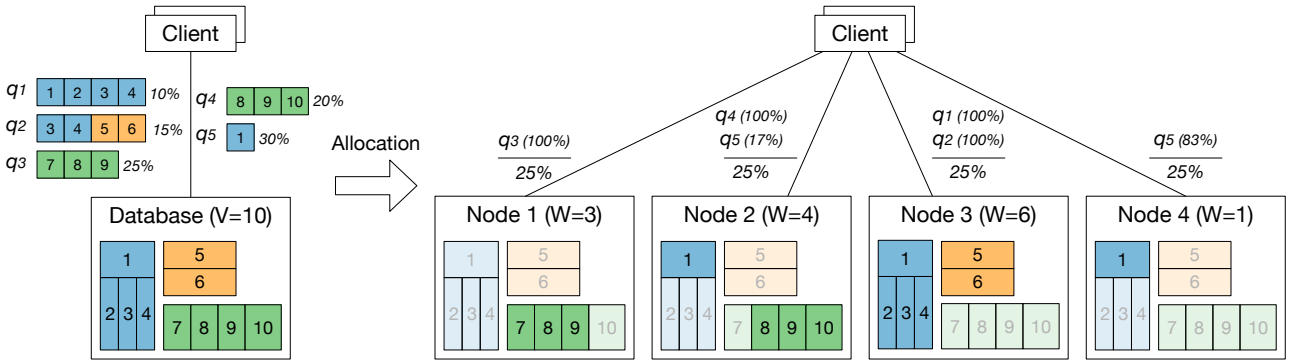

Figure 1: Workload-driven fragment allocation (based on [8]). The left-hand side of the figure visualizes the model input. The database consists of $N = 10$ fragments. $Q = 5$ queries correspond to different workload shares. Processing a query requires to store its accessed fragments. The objective is to minimize the overall memory consumption of the replication cluster while evenly balancing the load among $K = 4$ nodes. The right-hand side of the figure illustrates an optimal allocation with a total replication factor $W/V = 1.4$ and even workload distribution with 1/4 of the workload share assigned to each node.

3. RELATED WORK

Our workload distribution problem (see Section 2) is an allocation problem in the field of distributed database systems. Özsu and Valduriez [13] give an overview of related allocation problems. We summarize their overview as follows: (i) Constraints and optimization goals, e.g., performance, costs, and dependability, for allocation problems differ (see also [4]). (ii) Many formulations for allocation problems are proven to be NP-hard [5]. As a result, heuristics have to be used for large problem instances. (iii) As constraints and optimization goals differ, heuristics are often tied to specific formulations of allocation problems.

Our problem formulation is similar to the one presented by Rabl and Jacobsen [15]. We want to balance the load evenly to nodes to enable linear throughput scaling. Optimal solutions via LP do not scale, e.g., for the 22 TPC-H queries and using vertical partitioning with each of the 61 columns as an individual fragment, we were able to calculate optimal allocations for up to 8 nodes (termination after 8 h using a current laptop) [9].

To address large problem sizes, with thousands of queries, fragments, and dozens of nodes, Rabl and Jacobsen propose a greedy heuristic that assigns queries to nodes one after another. In specific, they order queries by the product of their workload share and the overall size of accessed fragments. Queries are then assigned to the node with the largest fragment overlap with already assigned queries. Nodes with no assigned queries are treated as if they have a complete overlap. If the load of the assigned query exceeds the capacity of a node, the query is inserted back to the assignment list with its remaining load.

We implemented Rabl and Jacobsen's algorithm and investigated the steps chosen during allocations for TPC-H and TPC-DS [8]. We made the following observation: Load capacities of nodes are often filled one after another, because nodes with many assigned queries are more likely to have high fragment overlap: Assigning a query to a node (may increase the number of allocated fragments and thus) increases the probability that the next query is assigned to the same node unless the node's load capacity is exhausted.

Our research addresses two shortcomings of Rabl and Jacobsen's approach: (1) When ordering queries and determining overlap, the algorithm does not consider the specific accessed fragments, but only their sizes. (2) When assigning queries to nodes, the remaining queries are not analyzed. In contrast, our decomposition approach (see Section 4) divides the problem into subproblems which preserves the structure of the problem. Specifically, we regard all queries (whose load we try to divide into workload chunks) with all individual fragments the queries access.

Solutions/allocations of our algorithm can be used for replicated databases, e.g., SAP HANA [12], Postgres-R [10] and in replication middleware [2, 3, 14]. Further, the calculated workload distributions support caching effects for systems like Amazon Aurora [17], which separate compute from storage.

Using LP to solve allocation problems is flexible compared to rule-based heuristics. We can add or change constraints, and modify the objective function to address varied allocation problems, e.g., requiring to store only a certain subset of fragments (instead of all) or demanding a similar memory consumption per node. Including such constraints into a rule-based heuristic is more challenging, because it is more difficult to decide how and in which part to adapt the algorithm without losing sight of the optimization goal.

4. DECOMPOSITION APPROACH

The complexity of our query-driven workload distribution problem increases with an increasing number of nodes, queries, and fragments. It is challenging to find good heuristics, as minimizing data redundancy and balancing the load on additional replicas are in conflict with each other.

The core idea of our decomposition heuristic is to split the workload iteratively using easier to solve but similar subproblems, forming a tree. In specific, we split the workload into chunks of queries, which access similar fragments. Thereby, we reduce the data redundancy in each splitting step. In the following, we describe the approach using an example. In [9], we present the corresponding LP model.

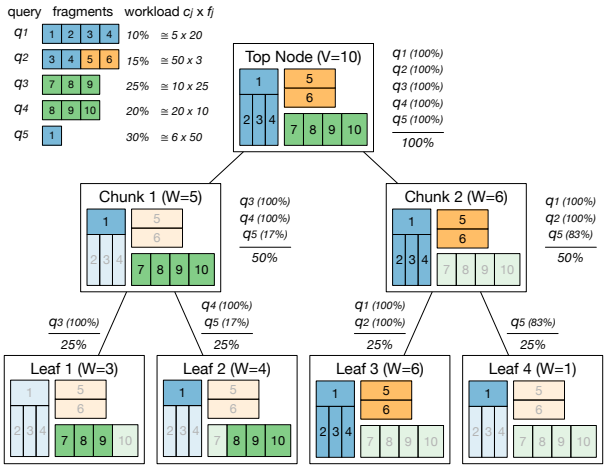

Figure 2: Decomposition approach (based on [9]). Iteratively splitting a workload for $K = 2 + 2$ nodes, leading to an even workload share of 1/4 at all leaves. The total replication factor is $W/V = 1.4$.

Figure 2 shows a decomposition of a workload, represented by the top node, to $K = 2 + 2 = 4$ final nodes, represented as leaves. Each parent node can have an arbitrary number of child nodes. The workload share of each child must correspond to the workload share of the leaves in its subtree. In Figure 2, the top node has two children. Both children have two leaves in their subtree. Therefore, their workload share is $2 * \frac{1}{4} = \frac{1}{2}$.

Using the decomposition approach, we can split the workload into an arbitrary number of final nodes with equal workload shares. Note, the existence of a solution with equal workload shares is guaranteed, because our model (like Rabl and Jacobsen's) allows to split query loads arbitrarily without regard to query frequencies and costs, which both are discrete (see [9, 15]).

With each decomposition, the problem gets easier as fewer relevant queries and accessed fragments have to be considered. For chunk 1 in Figure 2, the distribution problem is shrunken to $Q = 3$ queries and $N = 5$ fragments. By choosing the number of child nodes, we can control the problem complexity. For large problem instances, it is advisable to split the top node into a low number of chunks for lowest computation times. If the problem is still too large, other heuristic approaches, e.g., [15], can be used to split the workload close to the top node. Towards the leaves, our approach can be used.

4.1 Application to a Real-World Workload

In this section, we demonstrate our heuristic using a real-world workload. We analyzed queries against an accounting table, comprising of $N = 344$ columns/fragments. We extracted $Q = 4461$ query classes, whose query costs are distributed exponentially. In particular, 38 queries account for more than 95% of the workload share.

We compare our solution to the greedy heuristic by Rabl and Jacobsen [15]. Table 1 summarizes the results for $3 \leq K \leq 6$. Our solution reduces the replication factor in comparison to the greedy heuristic by 32-42%.

Table 1: Performance comparison: data replication factors of the rule-based heuristic by [15] (W^S) vs. our heuristic (W), see [9], with one decomposition level of B chunks and associated n_b nodes, $b = 1, ..., B$, summing up to K.

K	chunks B	nodes n_b	W/V	solve time	W/W^S
3	2	2+1	1.81	384 s	-32%
4	2	2+2	2.13	225 s	-42%
5	2	3+2	2.60	18 min	-33%
5	3	2+2+1	2.50	99 min	-36%
6	2	3+3	2.86	13 min	-37%

We used the Gurobi Optimizer [7] with a single thread. The computation times were between 2 min and 99 min, while focusing on reducing the replication factor. Different chunkings, e.g., $3+2$ and $2+2+1$, trade computation time for memory consumption. To reduce the computation time, we can also cluster fragments or queries, e.g., grouping the 4423 queries with lowest costs as a single query class with an aggregated workload share lower than 5%.

5. CURRENT AND FUTURE WORK

Numerical experiments show that our heuristic calculates allocations with lower memory consumption than the heuristic by Rabl and Jacobsen [15] for TPC-H [9] and a real-world workload (cf. Section 4.1). Currently, we conduct end-to-end evaluations by deploying the calculated fragment allocations in a PostgreSQL cluster and measuring the query throughput. In Section 5.1, we describe our preliminary insights.

In addition, we plan to investigate the impact of different database fragmentations and query clusterings on the allocation, especially, in the case of skewed model input (cf. Section 4.1). When running workloads in practice, fragment allocations and workload distributions must consider further factors, e.g., data modification costs, robustness against failures, or changing model inputs, that are not included in the basic allocation problem (cf. Section 2). We describe our plans to address these factors in the following sections.

5.1 End-to-End Evaluations

In practice, workloads consist of a number of active database connections sending a stream of queries. We model a workload as a set of queries in a potentially long period of time. The performance of an allocation in practice depends on the query timing, especially, whether all (or at least many) nodes are used for processing while there are pending queries. This can be supported by query scheduling: in case an incoming query is executable on multiple nodes, we send the query to the node with the lowest load.

Further, we can calculate allocations that trade memory consumption (allocating additional fragments to nodes) for flexibility (increased share of query loads to be processed by multiple replicas). Flexibility is also beneficial to handle imprecise query costs, e.g., caused by concurrency effects at runtime. Currently, we use average query execution times as query costs metric, because they are easy to obtain and widely applicable.

5.2 Data Modifications

As a result of inserts, updates, or deletes, data may change over time. We can adapt our basic model to include data

modification costs. We are able to model update costs either as execution costs similar to costs for read queries (see also [15]), or as fragment modification costs depending on the stored fragments per node, both in a linear way. Integrating modification costs for the decomposition heuristic is also possible. When splitting the workload into leaf nodes, we can include update costs precisely for node allocations. In upper levels, we may have to approximate update costs, because the exact number of leaf nodes for which update costs occur may be unknown.

5.3 Node Failures

Besides scalability, data replication enables high availability. Thereby, a database cluster can support different levels of robustness in case of node failures. Basic robust allocations ensure that each fragment is stored on multiple nodes, or queries can be processed on multiple nodes [15]. However, the workload distribution after node failures can be highly skewed. In future work, we will investigate allocations that allow an evenly balanced workload, even in the case of potential node failures.

5.4 Reallocation Costs

Workloads (including query costs) or fragment sizes may change. As a result, a current data allocation may not allow an even workload distribution anymore or a different allocation may reduce the memory consumption.

If workload changes are known in advance, Rabl and Jacobsen propose to calculate allocations for each scenario and merge all of the allocations to a combined allocation that is robust with regard to the workload changes [15]. An alternative approach may reallocate fragments to match the new workload. To avoid costly reallocations, we currently investigate how to take the current allocation into account in the algorithm. To avoid frequent reallocations, allocations should be robust with regard to minor workload changes (see Section 5.1).

5.5 Online Approach

As workloads and underlying data, and thus model inputs change, we want to adapt allocations over time, ideally without downtime or performance degradation. The previous sections described building blocks to implement a fault-tolerant and adaptive replication cluster. We can monitor the workload to detect when we have to adapt a current allocation. At that time we can calculate a new allocation, which is better suited to the current workload and which can be created with reasonable reallocation costs.

6. CONCLUSION

We presented the current status and plans to extend our work in the field of query-driven workload distribution and fragment allocation. While balancing the query load evenly, our decomposition approach calculates cluster configurations with lower memory footprint than state-of-the-art heuristics. We believe that our approach allows integrating further factors, such as data modification costs, robustness against node failures, and economical reallocations. Using LP, we can express these factors as constraints without changing the structure of the algorithm. In future work, we plan to not only extend our LP approach, but also demonstrate the extensions in end-to-end evaluations.

7. REFERENCES

[1] M. Boissier. Optimizing main memory utilization of columnar in-memory databases using data eviction. In *PhD@VLDB*, pages 1–6, 2014.

[2] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *SIGMOD*, pages 739–752, 2008.

[3] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *FREENIX@USENIX*, pages 9–18, 2004.

[4] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Comput. Surv.*, 14(2):287–313, 1982.

[5] K. P. Eswaran. Placement of records in a file and file allocation in a computer. In *IFIP*, pages 304–307, 1974.

[6] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, pages 173–182, 1996.

[7] Gurobi Optimization. `https://www.gurobi.com`.

[8] S. Halfpap and R. Schlosser. A comparison of allocation algorithms for partially replicated databases. In *ICDE*, pages 2008–2011, 2019.

[9] S. Halfpap and R. Schlosser. Workload-driven fragment allocation for partially replicated databases using linear programming. In *ICDE*, pages 1746–1749, 2019.

[10] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement database replication. In *VLDB*, pages 134–143, 2000.

[11] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB*, 5(1):61–72, 2011.

[12] J. Lee, S. Moon, K. H. Kim, D. H. Kim, S. K. Cha, W. Han, C. G. Park, H. J. Na, and J. Lee. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. *PVLDB*, 10(12):1598–1609, 2017.

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

[14] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.

[15] T. Rabl and H. Jacobsen. Query centric partitioning and allocation for partially replicated database systems. In *SIGMOD*, pages 315–330, 2017.

[16] D. Schwalb, J. Kossmann, M. Faust, S. Klauck, M. Uflacker, and H. Plattner. Hyrise-R: Scale-out and hot-standby through lazy master replication for enterprise applications. In *IMDM@VLDB*, pages 7:1–7:7, 2015.

[17] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD*, pages 1041–1052, 2017.