Query Centric Partitioning and Allocation for Partially Replicated Database Systems

Tilmann Rabl
University of Technology Berlin
Berlin, Germany
rabl@tu-berlin.de

Hans-Arno Jacobsen
University of Toronto
Toronto, Canada
jacobsen@eecg.toronto.edu

ABSTRACT

A key feature of database systems is to provide transparent access to stored data. In distributed database systems, this includes data allocation and fragmentation. Transparent access introduces data dependencies and increases system complexity and inter-process communication. Therefore, many developers are exchanging transparency for better scalability using sharding and similar techniques. However, explicitly managing data distribution and data flow requires a deep understanding of the distributed system and the data access, and it reduces the possibilities for optimizations.

To address this problem, we present an approach for efficient data allocation that features good scalability while keeping the data distribution transparent. We propose a workload-aware, query-centric, heterogeneity-aware analytical model. We formalize our approach and present an efficient allocation algorithm. The algorithm optimizes the partitioning and data layout for local query execution and balances the workload on homogeneous and heterogeneous systems according to the query history. In the evaluation, we demonstrate that our approach scales well in performance for OLTP- and OLAP-style workloads and reduces storage requirements significantly over replicated systems while guaranteeing configurable availability.

1. INTRODUCTION

Today, database systems are a core element of most data intensive systems. However, especially in smaller projects and startups, there is only a limited budget for data management. This is usually not an issue, as long as the database workload can be processed by a single server database system. With modern hardware, single-node database systems can be scaled up to enormous processing powers which come at enormous prices. The historically leading system in the TPC Benchmark™C of the Transaction Processing Performance Council has a total system cost of 30 million dollars[1]. This is not affordable for most startups, therefore, many projects use open source database management systems and off-the-shelf hardware. In these systems, the most difficult step is to transition from a single database server to a distributed database system.

As the amount of data produced is rapidly growing, shared-nothing architectures have emerged as a *de facto standard* [31, 55]. Affordable and massive hardware parallelism has led to new programming paradigms adopted in data processing such as MapReduce [22]. In general, these approaches exchange fundamental database system qualities for better scalability. Yet, systems with good scalability, frequently suffer from poor system resource utilization [36, 9]. The basis of success of relational database systems is their simple data model, combined with comprehensible, declarative data access. In distributed settings, this requires transparent access to the distributed data. Distributed access introduces dependencies, which have to be resolved automatically. Efficiency of a distributed system benefits most from global optimizations [50]. For a distributed database system, the data allocation or placement has major impact on overall performance.

There are several options for data allocation in a shared-nothing system. A popular approach is full replication [17, 46, 32, 7]. Full replication is frequently used in Web server setups or enterprise deployments[2]. Full replication has the advantage of good read scalability but suffers from low scalability for workloads with considerable amount of write access. This is due to the fact that updates have to be executed on all replicas. Furthermore, full replication is very space inefficient since the required storage space grows linearly in the number of nodes. Another alternative is declustering, common in parallel databases systems (e.g., NonStop SQL [26]), and key-value stores (e.g., Cassandra [35]). Declustering can scale well for high write access rates if the workload is balanced. The same is true for read access. In most systems, this is achieved by a randomized allocation in form of consistent hashing or round-robin or random partitioning. In general, this balances the workload, however, at the cost of data locality. For continuous data access patterns, randomized approaches incur additional network communication. An alternative approach, which can overcome both downsides is partial replication. In this approach, data is partitioned and replicated according to its access pattern. Data with high read access is replicated, while data with high write access is distributed. By analyzing the access patterns, a distribution and replication scheme can be found that maximizes the locality of the data and reduces the network communication while balancing the workload. In a shared-nothing environment, the overall system performance is strongly correlated with locality. This is because communication is expensive and will eventually always become the bottleneck [3].

In this paper, we present an allocation strategy that computes a (near) optimal solution to the allocation problem in shared-nothing and cluster databases. The allocation strategy results in a data layout that maximizes the throughput according to a theoretical maximum

[1]Top TPC-C Results - http://www.tpc.org/tpcc/results/tpcc_result_detail.asp?id=110120201

[2]A popular example is the Wikimedia Server setup – https://meta.wikimedia.org/wiki/Wikimedia_servers

SIGMOD'17, May 14 - 19, 2017, Chicago, IL, USA

ISBN 978-1-4503-4197-4/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3035918.3064052

and then minimizes the disk consumption and replication overhead. Based on a partitioning type and query access patterns, an allocation is computed that processes all queries with known access patterns locally and balances the workload across the nodes. It minimizes the overhead introduced by redundant updates, while balancing the workload and, thus, maximizing the overall throughput. We focus on deployments with a limited number of nodes, but do not restrict ourselves to this setting.

Our main contributions are the following: (1) We propose an analytical model for shared-nothing databases that allows us to calculate the theoretical speedup of a database system and enables the fully automated data allocation. (2) We present an automatic allocation strategy for homogeneous and heterogeneous deployments that maximizes the system's throughput and can be used for a wide area of database applications. (3) We analyze the performance of our solution with two industry standard benchmarks. Our evaluation shows that the proposed solution achieves perfect speedup for read-heavy workloads while reducing the storage requirements by 65%, and it achieves good speedup for write-heavy workloads, which outperforms full replication by a factor of 2.4. (4) We show the flexibility and elasticity of the our approach using real workload traces.

The rest of this paper is organized as follows: In the next section, we describe the cluster database system (CDBS) architecture and processing model. In Section 3, we present our allocation strategy in detail. Section 4 shows the performance evaluation and Section 5 describes the extensions we explored. In Section 6, we discuss related work. In the Appendix, we develop a complete example of the allocation algorithm, show the linear program formulation for optimal allocation, and give details on our k-safety extensions that ensure fault-tolerance in presence of failures of up to k backends.

2. CDBS PROCESSING MODEL

The architecture of a cluster database system is given in Figure 1. Its three-tiered design is a variant of the client-server model, in which the server component is further divided. Usually, the controller is a lightweight middleware system that distributes the incoming queries to the backends. The backends are completely independent database systems that each manage either a full replica or a part of the global database. The backend DBMSs do not share any resources. This design scales well in the number of nodes if single queries can be processed by a single backend. However, for large numbers of nodes, the controller can become a bottleneck. It is possible to use a hierarchy of controllers that manage subsets of backends. In this paper, we focus on single controller systems. To increase the speed of a

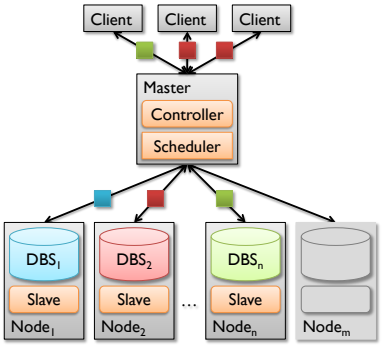


Figure 1: Cluster Database System Architecture

single DBMS, some form of parallelism has to be employed. Two types of processing a single query can be distinguished: intra-query parallelism and inter-query parallelism. A system with intra-query parallelism divides a single query into multiple query fragments and processes the query fragments in parallel. Using inter-query parallelism, multiple queries can be processed in parallel. Employing intra-query parallelism reduces the processing time of a single query, while inter-query parallelism increases throughput, while keeping the query processing time constant. Intra-query parallelism introduces dependencies between nodes, which leads to an increase in network communication. Apart from decision support and scientific database applications, queries are typically small, which means they can be processed in reasonable time on a single system. With the advent of multi-core processors, many DBMSs feature intra-query parallelism, which speeds up processing of a query on a single node, mitigating the requirement of intra-query parallelism at the cluster level.

In our CDBS model, we consider a query as an atomic unit. Each incoming query is entirely processed by a single backend. We must ensure that a backend has all input data for the queries it processes.[3] Since all queries can be processed locally, there is no need for communication between nodes. Each query can be sent to any backend that has all required data for the query. A common approach in cluster database systems is *full replication*, so each query can be processed by each backend. To achieve a balanced load distribution among nodes, a simple online strategy is applied for query scheduling: the *least pending request first* strategy sends a query to the backend, which at the time has the least queries queued [16]. This strategy is based on a greedy algorithm for online load balancing [12].

Updates cannot simply be processed by a single backend in all cases. They have to be executed on each replica of the updated data. The straightforward approach is, therefore, to send the update to every backend that holds data that is updated. Combined with the query processing approach above, this is called the *read once/write all* protocol (ROWA) [30]. In order to keep the database in a consistent state, it is important that all backends get the updates in the same order. Although there are more efficient approaches for update synchronization, such as primary copy [4] and lazy replication [34], we limit our discussion to the ROWA protocol in this paper due to space restrictions; other approaches could be easily incorporated into our model and system. In the following, we give estimations on the performance of a cluster database system using our processing model.

In its simplest form, a cluster database system places a full replica of the managed database on each backend. This means that each query can be sent to any backend. If the workload consists only of read requests, then the throughput of such a system increases linearly with the number of nodes.

The maximum throughput in a homogeneous system with only read requests is proportional to the number of nodes. If no additional overhead is introduced, it is equal to the number of nodes. However, updates have to be processed on each backend. Therefore, the throughput of the system decreases with the number of updates.

In a homogeneous, fully replicated system, each update incurs work on every node. Therefore, each update takes the same time as if the updates were processed on a single node, while reads (queries) can be distributed to any node. Thus, in theory, the processing time of updates is constant, while the processing time of reads is inversely proportional to the number of nodes. Hence, the speedup decreases with the processing time of updates. The correlation is defined by Amdahl's law [5]:

$$\text{speedup} = \frac{1}{\frac{\text{parallel}}{\#\text{nodes}} + \text{serial}} \tag{1}$$

[3]Note that an actual system may very well support distributed queries, while it is still aiming for local execution.

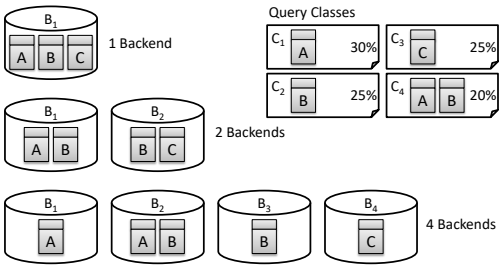


Figure 2: Read-Only Allocation on 1 to 4 Backends

In this context, *parallel* is the fraction of the query processing time and *serial* is the fraction of the update processing time. This model does not consider long running queries and deviations in the query processing time, which can be introduced by interdependencies of queries. Nevertheless, it enables a closed form expression for throughput prediction as can be seen in the evaluation in Section 4.

3. AUTOMATIC ALLOCATION

The primary optimization objective of our allocation strategy is maximizing throughput, while secondarily minimizing update and disk consumption overhead due to replication. The transformation into partitioned relations is calculated by the allocation algorithm directly based on the fragments and query classes. The algorithm can also be used with other partitioning schemes that are then treated as data fragments. Depending on the partitioning strategy used, the algorithm does not lead to balanced data allocation. The deployment of the allocation in the distributed system is done by cost optimal matching. The complete allocation process is a four step procedure: (1) query classification, (2) allocation calculation, (3) allocation improvement (optional), and (4) physical allocation.

In the first step, a query history or *journal* is analyzed; queries may be read or update requests. The analysis performs a classification of the queries. Queries are grouped according to the data they access. The classification determines the partitioning. If queries are grouped according to the tables they access, the allocation has no partitioning. If the queries are grouped according to the columns they access, the allocation has vertical partitioning. Finally, it is possible to group the queries based on their predicates and, thus, create a horizontal partitioning. It is also possible to use other partitioning strategies.

Based on the classification and on the set of nodes, the allocation is calculated. Each class of similar queries, i.e., each *query class*, is assigned to one or more backends, so that each backend has about the same amount of work in a homogeneous system and replication is reduced. The allocation does not guarantee a balanced data size, which is generally not achievable for a homogeneous workload assignment with minimized replication as can be seen in the examples below. The calculated allocation does not consider the existing data layout on the backends. To keep the overhead of the reallocation as small as possible, an optimal matching of the current and the calculated allocation is computed.

In the following, we give an example of our allocation strategy. The example shows a read-only database and an optimal allocation across different cluster sizes. In Appendix A, we provide a complete example of allocation with updates. In Figure 2, a database with three relations A, B and C is allocated with partial replication, but without partitioning. For sake of simplicity, the size of all relations is kept equal. The database is accessed with four types of read requests $C_1, \ldots, C_4$. The read request types abstract actual database queries. In this example, queries are classified by the relations they reference. The first type of queries references relation A and makes up 30% of the query workload. C_2 references relation B and makes up 25% of the workload, C_3 references relation C with 25% of the workload and C_4 references relations A and B, and it makes up 20% of the workload. The distributed database consists of 1 to 4 backends B_1,...,B_4, which all have equal processing power.

Three scales of a distributed database system can be seen. First, a single database backend B_1: It has to contain all three relations. For two backends, the workload can be evenly distributed. The query types C_1 and C_4 together make up 50% of the query load, as do C_2 and C_3. Hence, a possible solution is to allocate relations A to backend B_1 and relation C to backend B_2 and replicate B on both backends. With this allocation, both backends get an equal share of the workload and achieve the theoretical speedup of 2. It is easy to see that this configuration is optimal in terms of space efficiency. The load distribution is shown in the following table.

	C_1	C_2	C_3	C_4	Overall
B_1	30%	0%	0%	20%	50%
B_2	0%	25%	25%	0%	50%

The same speedup can be achieved in the read-only case by using full replication, however, with the exemplified allocation only one relation has to be replicated instead of all three relations. For four backends, each backend gets 25% of the workload. Query class C_1 has more than 25% of the workload and therefore it has to be assigned to more than one backend, so relation A is replicated on backends B_1 and B_2. C_2 and C_3 fit directly on a backend so they are allocated on B_3 and B_4, respectively. C_4 can be assigned to backend B_2 since it still has processing capacities. The resulting allocation has a theoretically optimal speedup of 4, while replicating only two tables. The load distribution is given below.

	C_1	C_2	C_3	C_4	Overall
B_1	25%	0%	0%	0%	25%
B_2	5%	0%	0%	20%	25%
B_3	0%	25%	0%	0%	25%
B_4	0%	5%	25%	0%	25%

We describe the individual steps of the allocation in detail below.

3.1 Classification

The basis of the classification is a query journal $\mathcal{J}$. The journal is a sequence of executed queries q. It does not need to contain every query executed, but it should be representative[4]. The domain of J is the set of all distinguishable queries Q, where two queries are distinguishable if they are not textually identical. Therefore, every $q \in Q$ may appear multiple times in $\mathcal{J}$. Since the order of elements is not important, we interpret $\mathcal{J}$ as a multiset with support Q and characteristic function j, i.e., the function j returns the number of occurrences of each query in the journal (we adopt the multiset formalism of [58]). Each query $q \in Q$ accesses a set of data fragments. Depending on the classification method, a data fragment f may be a relation, an attribute of a relation (or column) or a range or set of tuples (determined by predicates). Each query is classified to a single query class $C \in \mathcal{C}$, where the set of all query classes is a subset of the power set, $\mathcal{C} \subseteq \mathcal{P}(F)$ of all data fragments. The set of all data fragments can be the set of all tables (no partitioning), the set of all columns (vertical partitioning), a set of horizontal partitions, or a mixture of the above (hybrid partitioning). The granularity of the partitioning also determines the size of the data fragments. The

[4]For the sake of simplicity, we require a journal to contain queries of all occurring classes.

classification is defined formally as a function:

$$\text{classify}(q) = C, C = \{f \in F | q \text{ references } f\} \quad (2)$$

The *classify* function assigns a query q to a query class C, where C is the set of data fragments referenced by q. In the case of partial replication without partitioning these are the tables accessed by the query. If query classes are classified column-based, they contain a candidate key to ensure lossless reconstruction of all data. For simplicity, we presume this in the following. In the definition above, reads and updates are not differentiated. For update intensive workloads, the journal is divided into update requests and read requests. For an easier identification of read and update query classes, we denote read query classes with Q and update query classes with U. The classification is analogous to the definitions above. It results in two sets of query classes $\mathcal{C}_U$ and $\mathcal{C}_Q$:

$$\text{classify}(q) = C, C \in \begin{cases} \mathcal{C}_Q, & \text{if } q \text{ is read} \\ \mathcal{C}_U, & \text{else} \end{cases} \quad (3)$$

where $\mathcal{C}_Q \cup \mathcal{C}_U = \mathcal{C}$. Strictly speaking, $\mathcal{C}$ is a multiset, however, we treat it as a regular set here for simplicity. To calculate the allocation, the fraction of the overall workload produced by each query class, i.e., its *weight*, is needed. This can be calculated by summing up the execution times or a cost estimation (e.g., from the query optimizer as proposed in [43]). Using the following function, the relative weight of a query class is computed:

$$\text{weight}(C) = \frac{\sum_{q \in \{x \in Q | \text{classify}(x) = C\}} j(q) \cdot \text{weight}(q)}{\sum_{q \in Q} j(q) \cdot \text{weight}(q)} \quad (4)$$

The classification determines the partitioning calculated by the allocation algorithm. If all queries are classified to a single class, the resulting allocation is a full replication.

3.2 Allocation

Given a set of query classes and their *weight*, the task of the allocation is to distribute the data fragments in a way that queries can be answered locally by one of the backends. To ensure maximum efficiency, the allocation has to balance the load of all query classes across all backends and minimize replication in case of updates. A secondary goal of the allocation is to reduce overall data consumption. In this section, we define this form of allocation formally in a way that it can be directly translated into a linear program computing an optimal allocation. The formulation of the optimal allocation can be found in Appendix B. The allocation can be formally described as a function that assigns data fragments $f \in F$ to backend databases $B \in \mathcal{B}$. Because we aim for a balanced load, we distinguish backends by their query processing performance. This measure is given in relation to the sum of the query processing performances of all backends. Thus, the domain of the relative performance is $[0,1]$. Since the basis of our allocation is a classification, input to the allocation are the query classes $C \in \mathcal{C}$. Alternatively, an existing partitioning scheme could be given as input, e.g., [21, 45, 47]. The result is a partial replication. A partial replication can be represented as a multiset of backends. $\mathcal{P}(F) \times [0,1]$, A backend is a pair $\langle B^*, l \rangle$, comprised of a set of fragments $B^* \subseteq F$ and a maximum load (capacity) $l \in [0,1]$. The basis of the allocation for workloads with updates is a classification that distinguishes between update and read requests. Hence, input for the allocation are two sets of query classes, update query classes $\mathcal{C}_U$ and read query classes $\mathcal{C}_Q$, where $\mathcal{C} = \mathcal{C}_Q \cup \mathcal{C}_U$.

$$\text{allocation}(\mathcal{C}_Q, \mathcal{C}_U) = \mathcal{B}, \forall C \in \mathcal{C}, \exists B \in \mathcal{B}: f \in C \Rightarrow f \in \text{fragments}(B) \quad (5)$$

The function *fragments* specifies the set of fragments of a backend:

$$\text{fragments}(\langle B^*, l \rangle) = B^* \quad (6)$$

The allocation definition does not ensure that the load is balanced. In the case of read and update requests, a balanced load cannot generally be guaranteed for an optimal allocation. In a homogeneous environment, all backends obtain — if possible — the same share of the overall workload. In a heterogeneous environment, the shares of the workload that backends handle differ. To simplify the following equations, we introduce the function *load*, which specifies the relative performance of a backend:

$$\text{load}(\langle B^*, l \rangle) = l \text{ where } \sum_{B \in \mathcal{B}} \text{load(B)} = 1 \quad (7)$$

In a homogeneous cluster of s nodes, the relative load is $\frac{1}{s}$. To express the part of the weight of a query that is assigned to a backend, the *assign* function is used:

$$\text{assign}(C, B) > 0 \Rightarrow C \subseteq \text{fragments}(B) \quad (8)$$

An allocation that considers updates is *valid* if all query classes are allocated and if every update query class is allocated to all backends that store its referenced data. The following constraints must be satisfied:

$$\forall C \in \mathcal{C}_Q : \sum_{B \in \mathcal{B}} \text{assign}(C, B) = \text{weight}(C) \quad (9)$$

$$\forall C \in \mathcal{C}_U, \forall B \in \mathcal{B}, C \cap \text{fragments}(B) \neq \emptyset : \text{assign}(C, B) = \text{weight}(C) \quad (10)$$

$$\forall C \in \mathcal{C}_U : \sum_{B \in \mathcal{B}} \text{assign}(C, B) \geq \text{weight}(C) \quad (11)$$

If the first constraint is satisfied, all read queries are completely assigned. The second constraint guarantees that the update queries are assigned to every backend that contains referenced data. If the third constraint is satisfied, every update query is assigned to at least one backend. These constraints ensure that the allocation is valid. To ensure that the backends are as balanced as possible, first the functions *updates* and *updateWeight* are defined:

$$\text{updates}(C) = \{C_U \in \mathcal{C}_U | C \cap C_U \neq \emptyset\} \quad (12)$$

$$\text{updateWeight}(B, C) = \sum_{C_U \in \text{updates}(C)} \text{assign}(C_U, B) \quad (13)$$

The *updates* function specifies the set of update query classes that reference related data for a query class. The output of the *updateWeights* function is the sum of weights of update query classes that are already allocated to backend B, which have an overlapping data set to read query class C. The function is used to calculate the additional update load a read query class causes on a backend. To simplify the following equations, the sum of assigned workloads for a backend is defined as its *assignedLoad*:

$$\text{assignedLoad}(B) = \sum_{C \in \mathcal{C}} \text{assign}(C, B) \quad (14)$$

In a heterogeneous environment, the maximum load of each backend has to be considered. The sum of assigned workloads for an allocation considering updates with replicated update query classes is greater than 1. Hence, the maximum load for each backend must be scaled:

$$\text{scaledLoad}(B) = \text{load}(B) * \begin{cases} scale, & \text{if } scale > 1 \\ 1, & \text{else} \end{cases} \quad (15)$$

where $scale = max_{B' \in \mathcal{B}} \frac{\text{assignedLoad}(B')}{\text{load}(B')}$. For an optimal allocation in the sense of throughput, $scale$ must be minimized. This is ensured by the following constraint:

$$\begin{aligned} &\forall B, B' \in \mathcal{B}, B \neq B', \nexists C \in \mathcal{C}, \text{assign}(C,B) > 0: \\ &\text{scaledLoad}(B) - \text{assignedLoad}(B) \geq \\ &\quad \text{scaledLoad}(B') - \text{assignedLoad}(B') - \\ &\quad \text{updateWeigth}(B, C_Q) + \text{updateWeigth}(B', C_Q) \end{aligned} \tag{16}$$

This constraint ensures that no pair of backends differ in weight such that a query class could be shifted between them to balance the load. This does not guarantee a fully-balanced load, it only guarantees that the difference is as small as possible. The constraint is stricter than it would have to be for obtaining an optimized throughput. It would be enough to ensure that no query class could be shifted away from a backend B with $\text{scaledLoad}(B) = \text{assignedLoad}(B)$.

3.2.1 Maximum Speedup

As stated above, for a read-only workload, the theoretical speedup is always linear. In this section, we discuss the speedup for partially replicated allocations. The basis the general formulation of Amdahl's law (see Equation 1). To apply this law to the workload of a CDBS, the serial and parallel fractions of a query workload have to be identified. The read load can be parallelized completely. The update load can also be parallelized, by allocating unrelated query classes to different backends. However, single update queries have to be processed by every backend they are allocated to. Hence, the maximum speedup of a workload is bound by:

$$\text{speedup}_{\text{max}} \leq \frac{1}{\max_{C \in \mathcal{C}} \sum_{C_U \in \text{updates}(C)} \text{weight}(C_U)} \tag{17}$$

To calculate the speedup of a specified allocation, the serial part of the workload has to be specified. Since all query requests and update requests are processed in parallel, the serial part is in general always 0. However, if an update class is allocated to two backends, the updates that are part of the class have to be executed on both backends. Hence, the workload that the two backends can process is reduced by the weight of the update class. To allow all query classes to be processed, the overall workload is increased by at least the weight of the update class. Accordingly, the workload each backend has to process is increased. This corresponds to the *scaledLoad* as defined above. The *scaledLoad* of a backend is the *load* plus the backend's share of the additional *weight* of the replicated update classes. In a homogeneous environment, the load is $\frac{1}{|\mathcal{B}|}$ which corresponds to $\frac{1}{\#\text{nodes}}$. The *scaledLoad* is defined as $load \times scale$, where scale can be interpreted as the increased workload. Using this information, Equation 1 is modified to:

$$\begin{aligned} \text{speedup}_{hom} &= \frac{1}{\text{serial} + \frac{\text{parallel}}{\#\text{nodes}}} \\ &= \frac{1}{0 + \frac{scale}{\#\text{nodes}}} = \frac{1}{\text{scaledLoad}} \end{aligned} \tag{18}$$

In a heterogeneous environment, the *scaledLoad* is not equal for all backends. To obtain a meaningful measurement for the speedup, the average throughput per backend must be considered. Since the overall load is 1, the average load is $\frac{1}{|\mathcal{B}|}$. Using the $scale$ factor, the speedup in the heterogeneous environment is defined as:

$$\text{speedup} = \frac{1}{\frac{scale}{|\mathcal{B}|}} = \frac{|\mathcal{B}|}{scale} \tag{19}$$

Thus, an allocation with maximum speedup is found by minimizing the maximum *scaledLoad*, or, more generally, the $scale$ of an allocation. This is done using a linear program, a formulation directly derived from the above definitions can be found in Appendix B. Given the optimal $scale$, the replication can be minimized with a second linear program. In a read-only environment, the optimization of $scale$ is unnecessary since it always is 1.

3.3 Allocation Algorithm

The allocation problem formulated above is NP-hard and cannot be solved for realistic problem sizes [48]; therefore, a heuristic is needed. In the following, we present a greedy algorithm and a meta-heuristic algorithm. The allocation problem is similar to bin packing. Query classes with the highest weight and the largest data size, potentially create the most replication if scheduled late. Therefore, we use a first-fit strategy to calculate the allocation in polynomial time [19]. Input to the algorithm is the classification and an empty set of backends; output is the set of backends with allocated query classes. While this strategy does not consider previous allocations, the matching presented in Section 3.4 guarantees a cost minimal implementation. The complete algorithm is depicted in pseudo-code in Algorithm 1. The greedy algorithm starts by calculating the set $\mathcal{C}^*$:

$$\mathcal{C}^* = \mathcal{C}_Q \cup \{C_U \in \mathcal{C}_U | \nexists\, C_Q \in \mathcal{C}_Q : C_U \cap C_Q \neq \emptyset\} \tag{20}$$

It is the set of query classes that have to be assigned explicitly. This includes all members of $\mathcal{C}_Q$ as well as members of $\mathcal{C}_U$ that reference no data referenced by a read query. The members of $\mathcal{C}^*$ are sorted in descending order according to the product of the data size and the weight they impose on the backend; this includes the update query classes with overlapping data, which need to be allocated with the class. The result is stored in the sequence $\mathfrak{C}$ in Line 2. Then auxiliary variables for the current load of a backend, the scaled maximum load of a backend, and the unassigned weight of a query class are introduced in Lines 3 to 5.

For each query class in $\mathfrak{C}$, the weight that has to be assigned is stored. This is done in the while loop starting in Line 6. The query class which produces the most weight on a backend is allocated first. If all backends are already at their maximum capacity, their relative load is scaled. The difference to all backends is calculated in the foreach loop starting in Line 10. It is the size of the newly allocated data fragments or 0 if the backend is empty or ∞ if the backend is full.

Then, the query class is allocated to the backend with the least difference (Lines 17 to 32). First, the fragments are allocated to the backend (Line 18), then, the complete update load of the query class that is not yet allocated to the backend's load is added (Line 19). If the current query class is an update query class (Line 19), the query class is removed from the classes that need to be allocated, since further allocation of an update query class results in less throughput. If the current load of the backend is larger than its scaled load, *scaledLoad* is adapted. We omitted the adaption of the scaled load of the other backends; this is done according to Equation 15.

When the query class is a read query class (Line 24), the maximum load of the backend has to be scaled if the backend is already full or overloaded with the updates in question (Line 25). It is increased, such that the backend can hold a share matching its relative load. Then, two cases have to be considered: If the remaining weight of the query class does not fit on the backend (Line 27), as much weight of the query class as possible is assigned to the backend and the query class has to be assigned to other backends. If it does fit on the backend (Line 30), the remaining weight of the query class is added to the backend, the query class is processed entirely, and it is removed from the query classes that have to be allocated.

At the end of the while loop (Line 33), the sequence of query

Input: Classification $\mathcal{C}$, set of empty backends $\mathcal{B}$
Output: Heuristic allocation $\mathcal{B}$

1 $\mathcal{C}^* \leftarrow \mathcal{C}_Q \cup \{C_U \in \mathcal{C}_U | \nexists\, C_Q \in \mathcal{C}_Q : C_U \cap C_Q \neq \emptyset\}$;
2 $\mathfrak{C} \leftarrow$ sort $C \in \mathcal{C}^*$ descending to weight$(C \cup \text{updates}(C)) \times$ size$(C \cup \text{updates}(C))$;
3 currentLoad$(\mathcal{B}) \leftarrow 0$;
4 scaledLoad$(\mathcal{B}) \leftarrow$ load$(\mathcal{B})$;
5 restWeight$(\mathfrak{C}) \leftarrow$ weight$(\mathfrak{C})$;
6 **while** $C \in \mathfrak{C}$ **do**
7 **if** *all backends are full* **then**
8 **foreach** $B \in \mathcal{B}$ **do**
9 scaledLoad$(B) \leftarrow$ currentLoad(B) + load$(B) \cdot$ weight(C);
10 **foreach** $B \in \mathcal{B}$ **do**
11 **if** *currentLoad*$(B) =$ *scaledLoad*(B) **then**
12 difference$(C,B) \leftarrow \infty$;
13 **else if** *currentLoad*$(B) = 0$ **then**
14 difference$(C,B) \leftarrow 0$;
15 **else**
16 difference$(C,B) \leftarrow$ size$((C \cup \text{updates}(C)) \setminus \text{fragments}(B))$;
17 $B \leftarrow B \in \mathcal{B}$ with difference(C,B) minimal;
18 fragments$(B) \leftarrow$ fragments$(B) \cup C \cup$ updates(C);
19 currentLoad$(B) \leftarrow$ currentLoad(B) + weight(updates(C)) $-$ updateWeight(B,C);
20 **if** $C \in \mathcal{C}_U$ **then**
21 **if** *currentLoad*$(B) >$ *scaledLoad*(B) **then**
22 scaledLoad$(B) \leftarrow$ currentLoad(B);
23 $\mathfrak{C} \leftarrow \mathfrak{C} \setminus \{C\}$;
24 **else**
25 **if** *currentLoad*$(B) \geq$ *scaledLoad*(B) **then**
26 scaledLoad$(B) \leftarrow$ currentLoad(B) + load$(B) \cdot$ weight(C);
27 **if** *restWeight*$(C) >$ *scaledLoad*$(B) -$ *currentLoad*(B) **then**
28 restWeight$(C) \leftarrow$ restWeight$(C) -$ (scaledLoad$(B) -$ currentLoad(B));
29 currentLoad$(B) \leftarrow$ scaledLoad(B);
30 **else**
31 currentLoad$(B) \leftarrow$ currentLoad(B) + restWeight(C);
32 $\mathfrak{C} \leftarrow \mathfrak{C} \setminus \{C\}$;
33 sort$(\mathfrak{C})$ descending to restWeight and size;
34 **return** $\mathcal{B}$

Algorithm 1: Greedy Allocation Algorithm

classes is sorted again as in Line 2. When all query classes have been completely allocated, the algorithm terminates.

Meta Heuristic In general, the greedy heuristic computes a valid, but not an optimal solution. By altering the heuristically found solution, it is possible to generate a better solution. Since the search space is exponential, not all possible solutions can be tested. A common approach is to randomly generate valid mutations of the initial solution. To control the number and order of the mutations, a meta heuristic is used. Many meta heuristics follow a similar 5 step approach. (1) *Initialization*: an initial solution is generated randomly or deterministically; (2) *mutation*: a number of mutations is generated randomly; (3) *evaluation*: the mutations are evaluated according to a cost function; (4) *selection*: a new solution is selected from the mutations and the initial solution; *termination*: after a predetermined number of iterations or a defined stopping condition, the best solution is returned.

The algorithms differ mostly in the way in which the mutations are selected. An overview of different approaches can be found in [59].

For the allocation problem, an evolutionary programming approach was chosen. Evolutionary programming differs from other evolutionary approaches, such as genetic algorithms and evolution strategies, in that it does not recombine solutions [10]. Since we also use local improvements, the algorithm is classified asa hybrid heuristic or a memetic algorithm [39]. Evolutionary algorithms use a set of solutions that store the current population. For each iteration, a new population $\mathfrak{P}$ with a determined number of old solutions and mutations is generated. After a fixed number of iterations or a stopping condition, the best solution is returned. The pseudo-code is given in Algorithm 2. The algorithm starts with the generation of the initial

Input: Initial solution $S_{init} = (\mathfrak{A}, \mathfrak{L})$, size of the population p
Output: Optimized solutions $\mathcal{S}_{min}$

1 $\mathfrak{P} \leftarrow \{S_{init}\}$;
2 **for** *number of iterations* **do**
3 $\mathfrak{P}' \leftarrow$ mutate$(\mathfrak{P}, p)$;
4 $\mathfrak{P} \leftarrow$ select$(\mathfrak{P}, \frac{2}{3}$,best$) \cup$ select$(\mathfrak{P}', \frac{1}{3}$,best$)$;
5 $\mathfrak{I} \leftarrow$ select$(\mathfrak{P}, \frac{1}{3}$,random$)$;
6 $\mathfrak{P} \leftarrow \mathfrak{P} \setminus \mathfrak{I}$;
7 **foreach** $S \in \mathfrak{I}$ **do**
8 $S \leftarrow$ improve(S);
9 $\mathfrak{P} \leftarrow \mathfrak{P} \cup \mathfrak{I}$;
10 $\mathcal{S}_{min} \leftarrow \{S \in \mathfrak{P} | \text{weight}(S) = \min_{S' \in \mathfrak{P}} \text{weight}(S')\}$;
11 **return** $\mathcal{S}_{min}$

Algorithm 2: Evolutionary strategy

population in Line 1. In general, this can be a set of randomly generated allocations or simply a full replication. For a faster convergence of the algorithm, we start with the solution of the greedy heuristic. In the loop, in Line 2, the evolutionary process is executed, and the number of iterations determines the runtime. Another common stopping criterion is to stop if for a certain number of iterations, no better solutions have been found. However, this makes the runtime of the algorithm non-deterministic. The first step in the evolutionary approach is to mutate the population to generate the offspring (Line 3). In this step, a new set of valid allocations is generated by randomly altering the current population. As is common in evolutionary programming, the mutation is based on a single parent instead of combining parents. After that, the new population is chosen (Line 4). The strategy is a so-called $(\lambda + \mu)$ approach [53]. This means that parents and offspring are mixed for the new population, instead of only using the offspring. The function select(X, y, Θ) chooses a fraction y of the set X using the operator Θ. For the new population, the best $\frac{2}{3}$ of the old population and the best $\frac{1}{3}$ of the offspring survive (thus, ensuring convergence). In contrast to a classic evolutionary program, the memetic algorithm now chooses randomly $\frac{1}{3}$ of the new population that is improved using a local search (Lines 5 - 9).

We employ two local search strategies. The first strategy searches for backends with common pairs of allocated query and update classes. By shifting the query weight between backends, replicated update classes can potentially be omitted. The necessary constraints are shown below:

$$|\{C \in \mathcal{C}_Q | \text{assign}(C,B_1) > 0\} \cap \{C \in \mathcal{C}_Q | \text{assign}(C,B_2) > 0\}| \geq 2 \tag{21}$$

$$C_1 \neq C_2 \in \{C \in \mathcal{C}_Q | \text{assign}(C,B_1) > 0 \wedge \text{assign}(C,B_2) > 0\} : \text{updates}(C_1) \neq \text{updates}(C_2) \tag{22}$$

If these constraints hold, it is possible to reduce the number of allocated update query classes, by shifting the query classes and there-

fore potentially improve the throughput. This strategy can be applied to an allocation in $O(|\mathcal{Q}|^2 \times |\mathcal{B}|)$, since for each pair of query classes, all backends have to be analyzed. In some cases, where query classes are replicated, changing the distributed query class improves the allocation. The second strategy searches for pairs of update classes, where the replication of heavier update classes is reduced by increasing the replication of lighter update classes. The general formulation is:

$$C_{U1} \in \{C \in \mathcal{C}_U | \text{assign}(C,B_1) > 0 \wedge \text{assign}(C,B_2) > 0\} \quad (23)$$

$$C_{U2} \in \{C \in \mathcal{C}_U | \text{assign}(C,B_1) > 0\} : \text{weight}(C_{U2}) < \text{weight}(C_{U1}) \quad (24)$$

$$\sum_{\{C \in \mathcal{C}_Q | C \cap C_{U2} \neq \emptyset\}} \text{assign}(C,B_1) \geq \sum_{\{C \in \mathcal{C}_Q | C \cap C_{U1} \neq \emptyset\}} \text{assign}(C,B_2) \quad (25)$$

$$\sum_{C \in \bigcup_{\{C' \in \mathcal{C}_Q | \text{assign}(C',B_1) > 0 \wedge C' \cap C_{U2} \neq \emptyset\}} \text{updates}(C')} \text{weight}(C) < \text{weight}(C_{U1}) \quad (26)$$

The first constraint selects the replicated update class C_{U1}; it is an update class that is allocated to at least two backends (B_1 and B_2). The second constraint selects an update class C_{U2} on backend B_1 that has less weight than C_{U1}. The third constraint ensures that C_{U1} and the corresponding read query classes can be shifted completely to backend B_2. The last constraint ensures, that the replication does not increase the replicated updates due to other update classes that have to be replicated with C_{U2}. This improvement also lies in $O(|\mathcal{Q}|^2 \times |\mathcal{B}|)$, since for each pair of query classes, all backends have to be checked.

After a certain number of iterations, the best solutions, i.e., all solutions with minimal costs, are returned.

3.4 Physical Allocation

The allocation algorithm above calculates an allocation based only on the query history and the cluster environment. The algorithm does not take a previous allocation into account. Therefore, a new allocation has to be implemented on the existing database cost-efficiently. Several factors contribute to the cost of a physical allocation. Basically, it is an ETL process, i.e., data extraction, data transport, and data loading have to be considered. Data that is already allocated to a certain backend does not create any additional cost. However, data that has to be transferred to a new backend and imported into the database system does create noticeable cost. This cost is mainly related to the size of the data. Hence, a good approach is to reduce the amount of transferred data. In order to materialize the allocation in the system, a matching between the newly calculated allocation and currently installed allocation has to be found.

The problem can be modeled using a complete, weighted bipartite graph $G = (B' \cup B, E)$. Nodes B' represent the backends in the new allocation and nodes B the backends in the old allocation. Both node sets have the same size n. Each node in B' is connected with every node in B with an edge $e \in E$. The weight of an edge e_{vu} between node $B'_v \in B'$ and node $B_u \in B$ is the cost of allocating the data fragments in B'_v to B_u. Usually, this cost is mainly dependent on the size of the data that has to be transferred and imported. Hence, in many cases, a valid approximation of the weight is the sum of the sizes of data fragments which have to be additionally allocated to the backend. This can be calculated by the following equation:

$$\text{weight}(e_{uv}) = \sum_{f \in \text{fragments}(B'_v) \setminus \text{fragments}(B_u)} \text{size}(f) \quad (27)$$

Matching is a well studied problem; it is known as the assignment problem [13] which is a special form of linear optimization that is strongly polynomial. To generate a cost-minimal perfect matching, the Hungarian method is used [33, 40]. This algorithm calculates an optimal matching in $\mathcal{O}(n^3)$.

4. EVALUATION

We implemented a prototype to test the allocation algorithms. It consists of a controller that encapsulates all logic. The architecture can be seen in Figure 3. The controller has two modes of operation: allocation and query processing. In the query processing mode, the controller starts the driver that issues SQL requests. The requests are sent to the scheduler that holds a queue for each backend. The scheduler inserts incoming requests into the queues according to the *least pending request first* order. If the database is partially replicated across the backends, the scheduler also decides, which backend can handle a request. The data allocation is stored in the schema. For each queue, multiple connections are opened to the according database system and each connection holds a single request at a time. PostgreSQLand MySQLare used as backend database systems. Each processed request is stored in the query history along with its processing time. Furthermore, statistical data is stored in an embedded database for performance analysis. After a

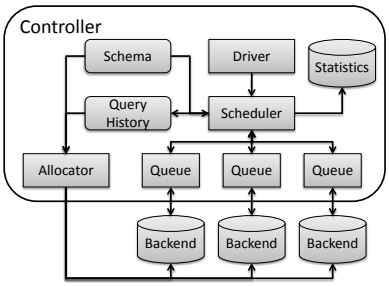
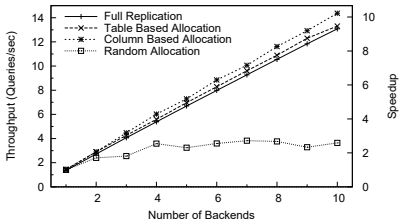


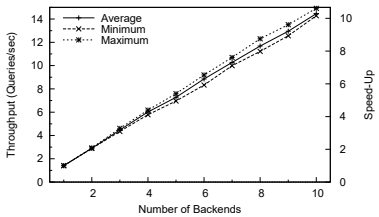
Figure 3: Architecture of the Prototype

test run, i.e., after a predefined number of requests, the controller changes to allocation mode. The allocator stops all backends, reads the query history and calculates an allocation according to the number of backends and the query history. Based on this allocation, the allocator assigns the data to the backends and starts them. Currently, full replication, table-based allocation, and column-based allocation are supported. The allocator waits until all backends have finished the bulk loading of the data. Then, the controller switches to query processing mode.

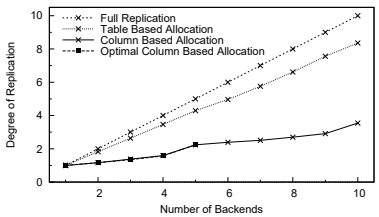
In order to guarantee comparable results, we do not further optimize the data layout. Therefore, the schema only includes indexes that are generated automatically (i.e., indexes on the primary keys). All tests are run on a 16 node high performance computing cluster. Each node has two Intel Xeon QuadCore processors with 2 GHz clock rate, 16 GB RAM and two 74 GB SATA hard disks with a RAID 0 configuration. We use separate nodes for the controller and the database backends. We tested our algorithms with TPC-H and TPC-App style benchmarks. For the scaling experiments with larger data sets, we used a 20 node virtualized cluster. The VMs are hosted on physical machines with each 2 Intel Xeon E5-2630 v3 CPUs, 128 GB RAM, and 2 600 GB SAS hard disks with RAID 1 configuration.



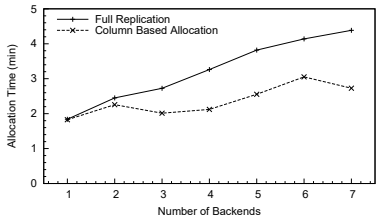
(a) TPC-H Throughput



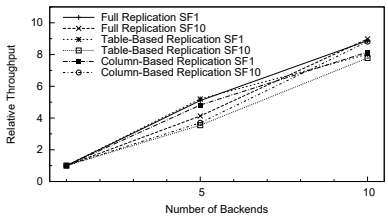
(b) TPC-H Throughput Deviation



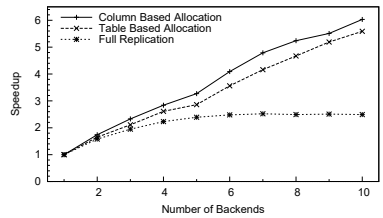
(c) TPC-H Degree of Replication



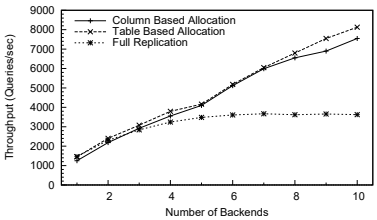
(d) TPC-H Duration of the Allocation



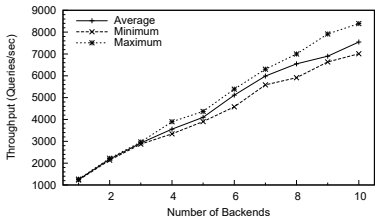
(e) TPC-H Scaling



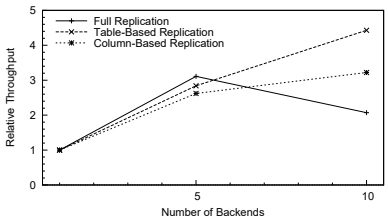
(f) TPC-APP Speedup



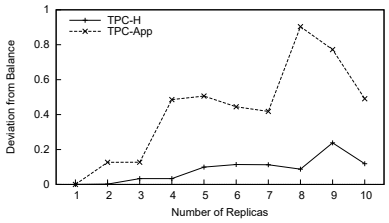
(g) TPC-App Throughput



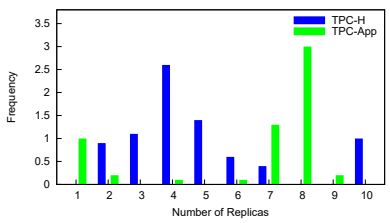
(h) TPC-App Throughput Deviation



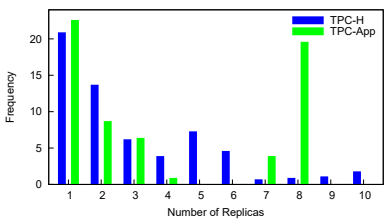
(i) TPC-App Large Scale



(j) Relative Load Balance TPC-H vs. -App



(k) Replication Histogram (Table-Based)



(l) Replication Histogram (Column-Based)

4.1 Read-Only Allocation Evaluation

Although TPC-H defines update statements, we used the benchmark without these to test our algorithm in a read only environment. PostgreSQL and MySQL have problems with the complexity of the queries in TPC-H. Since queries 17, 20, and 21 are disproportionately slow in PostgreSQL, we omitted them in the test. We used Scale Factor 1, which results in a 1 GB data set, since larger scale factors further limit the number of processable queries. We ran tests for full replication, table-based allocation, column-based allocation, and a random allocation. The random allocation randomly allocates column-based query classes on backends. Each test consisted of 10 runs on each number of backends. For the allocation test, we started with full replication for each scale in order to get an initial weight distribution for the queries. In each test, 10000 queries were sent to the database, created by the official TPC-H query generator. In Figure 4(a) the average of the runs can be seen.

It can be seen that all configurations scale linearly, except for the random placement. The random allocation throughput levels out at a speedup of 2.5, this is due to the imbalanced load. Both, the table and the column-based allocation, outperform the full replication. Since query classes differ considerably in their weight, some heavy classes are allocated exclusively to multiple backends. Because the backends are specialized on single query classes, less data is stored on the nodes and, hence, the caching on these backends improves. For the column-based allocation, the throughput further increases, since vertical partitioning improves the data transfer speed from the disk. The quality of the allocation is highly dependent on the quality of the classification and estimation of the weight of the query classes. In Figure 4(b) the minimum and maximum throughput of the column based allocation in the 10 test runs is shown. Although the column based allocation has the largest deviation in throughput, it is still never above 6%. This shows that the sum of the execution times of the queries is an excellent measure for the weight of a query class.

Table-based and column-based allocation reduce the amount of replication substantially. Figure 4(c) shows the degree of replication for full replication, table-based allocation, column-based allocation, and optimal column-based allocation. The degree of replication r for an allocation $\mathcal{B}$ is calculated as follows:

$$r(\mathcal{B}) = \frac{\sum_{B \in \mathcal{B}} \sum_{f \in B} \text{size}(f)}{\sum_{f \in F} \text{size}(f)} \tag{28}$$

The degree of replication for full replication matches the number of backends. The table-based allocation has a slightly reduced degree of replication. The data model in TPC-H is a data warehouse and nearly all queries reference the two big fact tables tables, which amount to 80% of the data. Therefore, it is not surprising that the table-based allocation uses over 80% of disk space of the full replication. Since the fact tables in TPC-H have many columns, column-based allocation leads to a considerable reduction of replication. For ten backends, the degree of replication is only 3.5. As a comparison, the result of an optimal allocation computed by a linear program is shown. Because of the high number of variables and constraints, the optimal allocation can only be calculated for up to 7 backends. It can be seen that the heuristically computed allocation is very close to the optimal allocation. For seven backends the difference in the degree of replication is 0.03. A positive side effect of the reduced degree of replication is an increased allocation speed. Figure 4(d) shows the duration of the allocation procedure for full replication and column based replication. The time measured includes the preparation of the table fragments, network transfer, and data load. Although full replication does not need the fragmentation, the reduced replication and thus reduced data transfer and load time outweigh the initial overhead.

To evaluate the behavior of the allocation for larger data sets, we run additional experiments with Scale Factors 3, 10, and 30. In Figure 4(e) the scaling behavior of the table based allocation and the column based allocation for TPC-H can be seen. The graphs show the relative performance of a 1, 5, and 10 backend setup for 1 and 10 GB results. Baseline is the performance of a single node with the same data set in all experiments. We increase the runtime of the experiments for larger setups in the order of the scale factor to reduce the impact of the different execution times at larger scale factors. As can be seen in the Figure, all allocation strategies show good scaling behavior with column-based allocation being as fast as full replication. Scale Factors 3 and 30 behave similarly.

4.2 Update-Sensitive Allocation Evaluation

To test the allocation on a workload with updates, we used a custom implementation of the TPC-App benchmark. TPC-App is a simulation of an online bookseller which is implemented using web services. The benchmark is scaled by increasing and decreasing the number of customers EB. We used $EB = 300$ in most experiments, which resulted in a database size of 280MB; additionally, we ran larger scale experiments with EB 12000, which resulted in 8GB of data. To test the allocation, the workflow of the web services was reimplemented and the queries were automatically generated. In the test, the number of queries was around 200,000 which we tested on full replication, table-based partitioning, and column-based partitioning. We tested the allocation on 1 to 10 backends, each test was repeated 10 times. The ratio of read to write queries was about 1 to 7. So for each read request 7 inserts and updates were sent. However, the select statements produced overall 3 times more workload than the updates. In particular, one complex read query class generated 50% of the workload although its queries made up only 1.5% of all queries. As mentioned above, a *read-once/write-all* strategy was used for the query scheduling. The workload consisted of 8 query classes for table-based allocation and 10 query classes for column-based allocation. All tables that are queried were also updated, therefore the column-based allocation always allocated the complete tables.

In Figure 4(f), the average speed up of all three allocation strategies can be seen. Due to the high write ratio, the full replication only reaches a speedup of 2.6, at which point it is stable. Additional backends do not reduce this speedup, but they also do not increase the throughput. Since the weight of the write query classes is 25% in total, this is not surprising. Using the formula presented in equation 1 the maximum theoretical speedup can be estimated:

$$speedup = \frac{1}{\frac{parallel}{\#backends} + serial} = \frac{1}{\frac{0.75}{10} + 0.25} = 3.07 \qquad (29)$$

The maximum speedup achieved by the full replication is 2.6, which is close to the theoretical maximum speedup. The table-based and the column-based allocation have a similar speedup that is not limited for the 10 backends. The speedup for the table-based and column-based allocation can be calculate in relation to the maximum weight of a write request class. In our implementation, the writes to the Order_Line table generate about 13% of the query weight; the maximum speedup can be reached if this write request class is allocated exclusively on a backend. For 10 backends, this results in an increase of the $scale$ factor to 1.3. Using Equation 19, we can compute the theoretical maximum throughput:

$$speedup = \frac{|B|}{scale} = \frac{10}{1.3} = 7.7 \qquad (30)$$

In our the tests, the maximum achieved speedup was 5.8 for table-based allocation and 6.7 for the column-based allocation. Both are close to the theoretical maximum speedup. The column-based allocation achieves a better speedup since it has a more query classes and, thus, has a more fine grained allocation. In Figure 4(g) the total throughput can be seen. The throughput is similar to the speedup. However, the column-based allocation is slightly slower than the table-based allocation and the full replication. This is due to some overhead in the query processing for the column-based allocation in our implementation. Another effect that can be seen is that for some cluster sizes the average throughput is slightly worse. This can also be seen in Figure 4(f), for 5 backends and 9 backends the speedup is slightly decreased. This shows that in some cases the allocation algorithm does not perform as well as in others. These are some corner cases where the algorithm tries to balance the load by allocating small parts of a query class to a backend, which result in an additional write overhead. Since column-based allocation has more query classes it is more vulnerable to misplacement. This can also be seen in Figure 4(h), it shows the deviation of the column based allocation. In contrast to the read only allocation, as shown in Figure 4(b), the read write allocation has a higher deviation. In Figure 4(i), speedup experiments for a large data set is shown. Here the update to read ratio was about 1 to 1, but the updates were more expensive due to the larger data size. This results in a reduced speedup of all allocation strategies, with full replication experiencing a slowdown for 10 nodes, while our allocation strategies exhibit good scaling behavior.

This difference between the read-only and the read-write allocation can also be seen in the graph in Figure 4(j). The graph shows the average balance of the column-based allocation of the TPC-H and the TPC-App workload. The balance is calculated as the relative deviation from the average overall processing time on all nodes. The presented values are the average balance of ten runs. As can be seen in the figure, the deviation increases with the number of backends and it is much less in the read-only case than in the read-write case. This is expected since the read-write case cannot be balanced in all cases in contrast to the read-only case and the individual query processing times are much more diverse in TPC-App than TPC-H. Although the deviation is almost 1 in some cases in the read-write case, the total throughput is not affected as much as one would expect. This is due to the fact, that in all cases the deviation stems from an underloaded node rather than an overloaded node. This means that most of the nodes have a good utilization while one node is un-

der utilized. An over utilized node that cannot be balanced by other nodes would have a much more dramatic effect on the throughput.

Figure 4(k) and Figure 4(l) show histograms of the replication of fragments for 10 nodes. This is again an average of 10 runs. In Figure 4(k) the frequency of replication numbers per table is shown for the TPC-H and TPC-App tests. It can be seen that in the case of TPC-H every table is replicated at least twice. The lineitem table is replicated on every node, since it is referenced in almost every query. The other tables are replicated less often, with most tables being replicated four times and each table at minimum twice. The TPC-App frequencies are very different. Here, one table that is heavily updated is always allocated on only one backend. The other tables are mostly read and thus are replicated. In the column-based allocation, there are many more possible fragments. In Figure 4(l), we show the frequency of replication number per column for both benchmarks, TPC-H and TPC-App. Interestingly, the histograms of both allocations are much more similar than in the table based case. This is due to the higher number of fragments and the algorithms effort to reduce replication. Since TPC-App has many queries that access the same columns, some columns are allocated more often in order to balance the load. Again, some of the columns are allocated on every node in TPC-H, but due to the vertical partitioning, the size of the allocated data is much less than in the table-based allocation.

Overall, both the read only and the update experiments show the effectiveness of our allocation algorithm. The read only allocation achieves an equal or higher throughput compared to the full replication, while decreasing disk usage. The update allocation greatly improves the system throughput while also reducing disk usage.

5. EXTENSIONS

Our allocation scheme can be extended to ensure k-safety to achieve high availability, meaning that k systems can fail without data loss and service interruption. To do so, we ensure that each query class is allocated at least $k + 1$ times. The adapted constraints and the heuristic algorithm can be found in Appendix C.

The processing model presented in Section 2 is fairly robust to changes in the workload. The allocation presented further supports this robustness in many cases. We can consider four cases of workload changes, an increase or decrease of weight of a certain query or update class. In the case of update classes, the allocation aims at minimal replication, which will be favorable in cases of increased update weight in contrast to a solution with higher replication. In the case of lower update weight, a minimal replication is still not harmful for the throughput. In the case of an increase of weight of a query class, a backend will experience higher loads, while a reduction of the weight of a query class will result in lower load. The reduction of performance through over-utilization can be estimated according to Formula 19, where $scale$ is the factor of over-utilization of a node. Consider the example of 4 backends in Figure 2, if the weight of Query Class C is increased to 27%, the maximum achievable speedup is reduced by to 3.7 instead of 4. This is, however, the worst case since C is the only class allocated on B_4. In general, there is some flexibility for the scheduling of query classes due to the remaining replication and co-allocation of query classes.

It is also possible to further increase the robustness of an allocation by ensuring that for each query class a certain percentage of change can be tolerated. An allocation can tolerate a change of the workload by shifting weights between backends in case of replicated query classes without loss of performance. If each backend contains query classes that can be (partially) shifted to another backend, the total allocation is robust. In the algorithm this can be implemented by a check for this condition for each fully loaded backend. If the requirement is not met, a new set of query classes with no weight

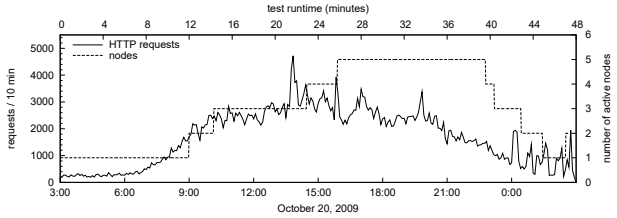


Figure 4: Number of Active Servers Compared to Workload

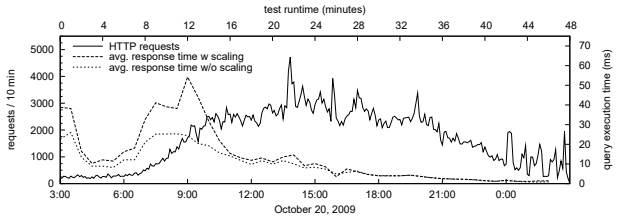


Figure 5: Average Response Time Compared to Workload

is added to the allocation, which combined weight amounts to the required percentage of robustness.

Using matching, a cost optimal scaling of the database system can be calculated. If a system is scaled, a new allocation is calculated and either new nodes are added or removed. The data mapping is modeled using a complete bipartite graph $G = (B' \cup B, E)$. In both situations, the cardinality of B' and B is different: For an increase in the number of nodes, $|B'| > |B|$, and for a decrease in the number of nodes, $|B'| < |B|$. In order to use the Hungarian method, both sets have to have the same number of nodes. For a scale out, the old allocation is simply increased with empty virtual backends. These present the new, unpopulated nodes. If the system is scaled down, it is reasonable to remove backends, which have less disk capacity or bad connectivity. In this case, the backends concerned are removed and the matching is done as if there was no scaling. If the backends to be removed can be freely chosen from all backends, the new allocation is extended with empty backends and the matching is performed. The backends of the old allocation that are matched with the empty backends are then removed from the cluster. We have implemented an autonomic CDBS using a simple scaling logic. Using real workload trace and the TPC-App benchmark, we tested autonomic scaling. The workload trace is taken from the backend database accesses of a Web-based e-learning tool. Due to privacy restrictions, we could only use statistics, not actual queries and data. The system was scaled up and down based on the average response time of the queries. In Figure 4, the original request profile and the number of active nodes during the test are shown. The requests are scaled up by a factor of 40 to increase the throughput, the maximum load was 250 queries per second. As shown in Figure 5, the autonomic system has a slightly increased response time, however, the average response time never exceeds 50 milliseconds and is 10 milliseconds on average. Most importantly, the throughput of the system is never decreased as compared to a system with static maximum size. We treat changing workloads similar to autonomic scaling. Two different cases are considered, predictably changing workloads and fundamentally changing workloads. The former can be in the form of periodic changes such as daily patterns or simple fluctuations in the workload. The latter is a result of changes in user behavior, user interface, and application logic. Fundamental workload changes are detected through permanent, non-optimal backend utilizations that then trigger reallocation. For periodic and fluctuating workloads, a reallocation is inefficient if the allocation cost outweighs the perfor-

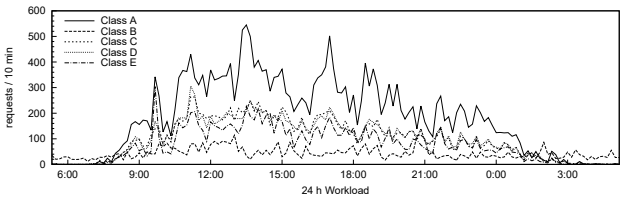


Figure 6: Distribution of Query Classes over a Day

mance benefits. To address this issue, the allocation needs to account for different workload distributions. Consider the workload in Figure 6, which shows the most frequently accessed query classes from the above workload. As can be seen in the figure, the ratio of accesses between the query classes changes throughout the day. Most notably, at night (3 am to 8 am), Class B is dominating, while it has lowest weight during the day. To directly treat this variance, we segment the query history into multiple parts using a one hour sliding window to compare variances. In the example, this leads to 4 segments (3 am to 8:30 am, 8:30 am to 10:30 am, 10:30 to 10:30 pm, 10:30 pm to 3 am). For each of these segments, we calculate a separate allocation. Using the Hungarian method, we merge all of the allocations to a combined allocation that is robust to the changes in the workload.

6. RELATED WORK

There are many implementations of cluster database systems that use the processing model presented above. The C-JDBC project is a middleware platform for transparent, table-based database replication [17]. The Ganymed project provides a middleware layer that allows replication of databases and transparent access [46]. The system is divided into a master and several satellite database systems, in which the satellites feature read-only access, while write access is processed only on the master node. The system Middle-R implements a middleware for database replication [38, 44]. The work is based on the Postgres-R system [32], which implemented a similar system within the PostgreSQL DBMS. The system uses group communication to synchronize update requests. Closest to our approach is C-JDBC. For full replication, the processing model is identical. Ganymed follows an equivalent approach for query processing, however, in contrast to our work, it uses a primary copy update mechanism. The same is true for MIDDLE-R; while query processing is equivalent to the CDBS model, updates are synchronized via group communication. This reduces the network traffic, but again limits predictability.

An allocation algorithm can have multiple goals, such as performance optimization, reliability, etc. In [42] a general problem definition is presented, which respects various factors such as network characteristics and properties of the host system. The resulting optimization problem is NP-hard and therefore not feasible for realistic problem sizes. There have been many modifications of the model, for example, for sites in communication networks [8] and for special purposes such as heterogeneous storage [15, 11]. To compute an allocation for realistic problem sizes, heuristics are used. These either exploit the relation to the bin packing problem and apply first-fit and best-fit strategies [14, 51], or use meta-heuristics such as simulated annealing [6] or evolutionary strategies [2].

Our allocation approach is also presented as an optimization problem. In contrast to other allocation strategies (e.g., [42, 8]), we do not consider network traffic, since we aim at local execution and consider cluster hardware configurations rather than wide area networks. Because of the NP-completeness, we use a best-fit strategy to find an initial solution and an evolutionary algorithm to improve the initial solution. However, the independent calculation of partitioning and allocation is in general inferior to an integrated approach, since an optimal partitioning can only be determined considering the optimal allocation of partitions [18]. In the following, we present a few integrated allocation strategies, all of which consist of at least a partitioning and an allocation part.

In [20], the database system Bubba was presented which uses a partial declustering and a greedy heuristic for data placement. Its partitioning approach is a simple range-based declustering that fragments each relation into a fixed number of equal-sized partitions. These partitions are allocated according to their heat (i.e., the access frequency) so that each node in the system has an equal heat. The placement strategy is a simple first-fit approach, which does not guarantee a balanced load. If the systems load is unbalanced, the placement strategy is applied to the fragments with the highest temperature (i.e., heat divided by size); this way data movement is minimized. Although Bubba was built as a parallel database system, it exploited data locality similar to our system, in contrast to other parallel database systems [52, 23] or multi-disk systems [1]. It uses no workload-aware partitioning and uses access frequencies rather than actual costs. A workload-aware approach was presented in [41]. It uses a linear programming approach to balance the workload. This approach, however, does not consider locality.

A purely dynamic approach to the allocation problem is presented in [27]. The data is range-partitioned into equal sized fragments and allocated to different nodes. The approach makes no assumption about the initial allocation. The optimization goal is storage balance, although the authors claim that the approach can be generalized to load balance. Whenever the storage on the nodes is out off-balance, according to a given threshold, one of two rebalancing strategies is invoked: the first strategy shifts weight to a node containing the neighbor fragment, while the second strategy shares weight with an empty node. Although the algorithm aims at parallel database systems, the general idea can be adapted for cluster database systems.

A decentralized fragmentation approach was presented in [29]. DYFRAM is similar to the above approach, a purely dynamic approach, that makes no assumption about the initial allocation and partitioning. Based on statistics captured for each tuple, the number of local accesses versus the number of remote accesses is calculated. The following strategies are employed: (i) if a fragment is often requested remotely, a copy is placed on the remote site; (ii) if access frequencies within a fragment vary significantly locally and remotely, the fragment is further fragmented; (iii) if a fragment is requested mostly from a remote site, it is moved. The approach aims to localize the data access in a distributed system with heterogeneous access to data, which is common in federated database systems. A similar approach was presented in [57]. This differs from our problem statement, which aims at load balancing. Also, it is questionable whether tuple-based statistics are manageable for larger databases.

In [25], the Tashkent+ system is presented, a successor of the Tashkent system [24]. The main target of the Tashkent+ system is load balancing. The basic assumption is that all transactions are known in advance. Tashkent+ schedules transactions on nodes such that they can be processed in memory. Starting from a fully replicated allocation, the backends are specialized to certain transaction types. Unused relations are eventually dropped and hence an optimized data layout is generated. Due to the memory-aware scheduling, a super-linear speedup is achieved. However, no partitioning is used. The approach has a similar processing model to our approach. A recent partitioning scheme for heterogeneous distributed setups was presented by Li et al. [37]. The authors use a linear programming model to solve the partitioning and allocation problem.

Unlike our algorithm, this approach targets intra-query parallelism and optimizes the individual query performance, while our approach optimizes the total system throughput. In recent work, Zamanian et al. presented a horizontal partitioning scheme for the XDB cluster database systems [60]. This work considers key relations and workloads to co-allocate horizontally partitioned tables; this approach is orthogonal to our work and could be integrated into our approach.

7. CONCLUSIONS

In this paper, we presented a processing model for a cluster database architecture, a common architecture for distributed database systems. Our model allows for an accurate analysis of the requirements of a distributed database system. Many research projects use very complex models to obtain an exact view of query processing in distributed system. These models, however, limit the model application to small problem sizes or controlled environments. In contrast, our model allows for the autonomous determination of all parameters. Therefore, all algorithms based on our model are fully implemented. This increases the possibilities for self-management of the database. We demonstrated this by autonomically scaling our system.

To increase the throughput and scalability of a CDBS, we presented a formal definition of the allocation problem in cluster database systems. As for the processing model, the allocation problem is reduced to the necessary parameters, which can be computed autonomically. We have shown an optimal formulation and a heuristic allocation algorithm that optimize the storage efficiency of the cluster database system architecture; as additional benefit, query processing performance is increased. Our evaluation shows that for read-only workloads, our algorithm computes allocations that reduce the storage requirements by 65% and achieve super-linear speedup. For workloads with updates, our algorithm increases performance by up to 2.4 times, as compared to a fully replicated system.

8. ACKNOWLEDGMENTS

This work has been supported through grants by the German Ministry for Education and Research as Berlin Big Data Center (funding mark 01IS14013A), through grants by the European Union's Horizon 2020 research and innovation program under grant agreement 688191, as well as through grants from Natural Sciences and Engineering Research Council of Canada and the Ontario Research Fund.

9. REFERENCES

[1] R. Agrawal, S. Chaudhuri, A. Das, and V. R. Narasayya. Automating Layout of Relational Databases. In *ICDE*, pages 607–618, 2003.

[2] I. Ahmad, K. Karlapalem, Y.-K. Kwok, and S.-K. So. Evolutionary Algorithms for Allocating Data in Distributed Database Systems. *Distributed and Parallel Databases*, 11(1):5–32, 2002.

[3] A. Alba, V. Bhagwan, M. Ching, A. Cozzi, R. Desai, D. Gruhl, K. Haas, L. Kato, J. Kusnitz, B. Langston, F. Nagy, L. Nguyen, J. Pieper, S. Srinivasan, A. Stuart, and R. Tang. A Funny Thing Happened on the Way to a Billion... *IEEE Data Engineering Bulletin*, 31(4):27–36, 2006.

[4] P. A. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. In *ICSE*, pages 562–570, 1976.

[5] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS*, pages 483–485, 1967.

[6] R. R. Amossen. Vertical Partitioning of Relational OLTP Databases Using Integer Programming. In *ICDEW*, pages 93–98, 2010.

[7] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed Versioning: Consistent Replication for Scaling Back-End Databases of Dynamic Content Web Sites. In *Middleware 2003*, pages 282–304. Springer Berlin Heidelberg, 2003.

[8] P. M. G. Apers. Data Allocation in Distributed Database Systems. *ACM Transactions on Database Systems*, 13(3):263–304, 1988.

[9] L. A. Barroso and U. Hölzle. The Case for Engergy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007.

[10] H.-G. Beyer. *Theory of Evolution Strategies*. Springer Berlin / Heidelberg, 2001.

[11] B. Bhattacharjee, M. Canim, C. A. Lang, G. A. Mihaila, and K. A. Ross. Storage Class Memory Aware Data Management. *IEEE Data Engineering Bulletin*, 33(4):35–40, 2010.

[12] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[13] R. Burkard, M. Dell'Amico, and S. Martello. *Assignment Problems*. Society for Industrial and Applied Mathematics, 2009.

[14] B. Calder, C. Krintz, S. John, and T. Austin. Cache-Conscious Data Placement. *SIGOPS Operation System Review*, 32(5):139–149, 1998.

[15] M. Canim, B. Bhattacharjee, G. A. Mihaila, C. A. Lang, and K. A. Ross. An Object Placement Advisor for DB2 Using Solid State Storage. *PVLDB*, 2(2):1318–1329, 2009.

[16] E. Cecchet. RAIDb: Redundant Array of Inexpensive Databases. In *ISPA*, pages 115–125, 2004.

[17] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *USENIX*, 2004.

[18] S. A. Ceri, M. Negri, and G. Pelagatti. Horizontal Data Partitioning in Database Design. In *SIGMOD*, pages 128–136, 1982.

[19] E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation Algorithms for Bin Packing: A Survey. In D. S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 2. PWS Publishing Company, 1996.

[20] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data Placement in Bubba. *SIGMOD Record*, 17(3):99–108, 1988.

[21] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1-2):48–57, 2010.

[22] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 51(1):107–113, 2008.

[23] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *VLDB*, pages 228–237, 1986.

[24] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication. In *SIGOPS/EuroSys*, pages 117–130, 2006.

[25] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-Aware Load Balancing and Update Filtering in Replicated Databases. In *EuroSys*, pages 399–412, 2007.

[26] S. Englert, J. Gray, T. Kocher, and P. Shah. A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases. Technical Report 89.4, Tandem Computers Inc., 1989.

[27] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems. In *VLDB*, pages 444–455. VLDB, 2004.

[28] J. Gray. Why Do Computers Stop and What Can be Done About It? Technical Report 85.7, Tandem Computers, 1985.

[29] J. O. Hauglid, N. H. Ryeng, and K. Nørvåg. DYFRAM: Dynamic Fragmentation and Replica Management in Distributed Database Systems. *Distributed and Parallel Databases*, 28(2):157–185, 2010.

[30] A. A. Helal, A. A. Heddaya, and B. B. Bhargava. *Replication Techniques in Distributed Systems*. Springer, 1996.

[31] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[32] 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.

[33] H. W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistic*, 52(1):7–21, 2005.

[34] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10:360–391, 1992.

[35] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[36] J. Leverich and C. Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. In *HotPower*, New York, NY, USA, 2009. ACM.

[37] J. Li, J. Naughton, and R. V. Nehme. Resource Bricolage for Parallel Database Systems. *PVLDB*, 8:25–36, 2014.

[38] J. M. Milan-Franco, R. Jiménez-Peris, M. P. no Martínez, and B. Kemme. Adaptive Middleware for Data Replication. In *Middleware*, pages 175–194, 2004.

[39] P. Moscato. On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms. Technical Report Caltech Concurrent Computation Program 158-79, California Institute of Technology, Pasadena, CA, USA, 1989.

[40] J. Munkres. Algorithms for the Assignment and Transportation Problems. *Journal of the Society of Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[41] O. Ozmen, K. Salem, J. Schindler, and S. Daniel. Workload-Aware Storage Layout for Database Systems. In *SIGMOD*, pages 939–950, 2010.

[42] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer New York, Upper Saddle River, NJ, USA, 2011.

[43] S. Papadomanolakis, D. Dash, and A. Ailamaki. Efficient Use of the Query Optimizer for Automated Physical Design. In *VLDB*, pages 1093–1104. VLDB Endowment, 2007.

[44] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems*, 23(4):375–423, 2005.

[45] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *SIGMOD*, 2012.

[46] C. Plattner, G. Alonso, and M. T. Özsu. Extending DBMSs with Satellite Databases. *VLDBJ*, 17(4):657–682, 2008.

[47] A. Quamar, K. A. Kumar, and A. Deshpande. SWORD: Scalable Workload-aware Data Placement for Transactional Workloads. In *EDBT*, 2013.

[48] T. Rabl. *Efficiency in Cluster Database Systems*. PhD thesis, University of Passau, 2011.

[49] P. Rubel, M. Gillen, J. Loyall, R. Schantz, A. Gokhale, J. Balasubramanian, A. Paulos, and P. Narasimhan. Fault Tolerant Approaches for Distributed Real-time and Embedded Systems. In *MILCOM*, pages 1–8. IEEE, 2007.

[50] C. Rusu, A. Ferreira, C. Scordino, and A. Watson. Energy-Efficient Real-Time Heterogeneous Server Clusters. In *RTAS*, pages 418–428, 2006.

[51] D. Sacca and G. Wiederhold. Database Partitioning in a Cluster of Processors. *ACM Transactions on Database Systems*, 10(1):29–56, 1985.

[52] P. Scheuermann, G. Weikum, and P. Zabback. Data Partitioning and Load Balancing in Parallel Disk Systems. *VLDBJ*, 7(1):48–66, 1998.

[53] H.-P. Schwefel and G. Rudolph. Contemporary Evolution Strategies. In *Proceedings of the Third European Conference on Advances in Artificial Life*, volume 929 of *Lecture Notes in Computer Science*, pages 893–907, Berlin, Germany, 1995. Springer.

[54] S. Shankland. Google spotlights data center inner workings, May 2008. https://www.cnet.com/news/google-spotlights-data-center-inner-workings/.

[55] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and Parallel DBMSs: Friends or Foes? *CACM*, 53(1):64–71, 2010.

[56] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, pages 553–564. VLDB Endowment, 2005.

[57] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A Wide-Area Distributed Database System. *VLDBJ*, 5(1):48–63, 1996.

[58] A. Syropoulos. Mathematics of Multisets. In *WMP*, volume 2235 of *Lecture Notes in Computer Science*, pages 347–358, Berlin, Germany, 2001. Springer.

[59] S. Voß. Meta-heuristics: The State of the Art. In *ECAI*, pages 1–23, 2001.

[60] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD*, pages 17–30, 2015.

APPENDIX

A. COMPLETE EXAMPLE

Consider the heterogeneous allocation example in Figure 7. It contains 7 query classes: 4 reads and 3 updates. The backends have different processing powers, while B_1 and B_2, each process 30%, B_3 and B_4 each process 20%. For the sake of simplicity, we perform table-based classification and, thus, no partitioning and consider each table to have the same size of 1 unit.

First, the set of query classes, $\mathcal{C}^*$, that have to be allocated explicitly is calculated. Since all fragments in update classes are also referenced in read classes, only read classes have to be explicitly allocated. Hence, $\mathcal{C}^* = \{Q_1, Q_2, Q_3, Q_4\}$. This set is sorted according to the weight it will produce on a backend multiplied with the size of all tables that need to be allocated. This is the weight of the query class itself and all referenced updates. For Q_1, this is $\text{weight}(Q_1) + \text{weight}(U_1) = 28\%$. This is multiplied with the size of tables, resulting in the following list: $\mathfrak{C} = (Q_4, Q_2, Q_1, Q_3)$. While Q_2 and Q_4 have the same weight to be allocated, Q_4 has two tables and, thus, size 2. The algorithm uses the auxiliary lists *currentLoad*, *scaledLoad*, and *restWeight*. Their initial status is as follows:

$$\text{currentLoad} = (B_1 : 0, B_2 : 0, B_3 : 0, B_4 : 0) \tag{31}$$

$$\text{scaledLoad} = (B_1 : 0.3, B_2 : 0.3, B_3 : 0.2, B_4 : 0.2) \tag{32}$$

$$\text{restWeight} = (Q_1 : 0.24, Q_2 : 0.2, Q_3 : 0.2, Q_4 : 0.16) \tag{33}$$

In the main loop, query class Q_4, which initially is in $\mathfrak{C}$, is processed first. The difference to each backend is calculated, which is 0 for all backends, since all are empty. In the next step, all fragments of Q_4 and its updates (i.e., U_1 and U_2) are allocated to B_1. The resulting allocation matrix is:

	A	B	C
B_1	1	1	0
B_2	0	0	0
B_3	0	0	0
B_4	0	0	0

The load matrix is:

	Q_1	Q_2	Q_3	Q_4	U_1	U_2	U_3	Overall
B_1	0%	0%	0%	0%	4%	10%	0%	14%
B_2	0%	0%	0%	0%	0%	0%	0%	0%
B_3	0%	0%	0%	0%	0%	0%	0%	0%
B_4	0%	0%	0%	0%	0%	0%	0%	0%

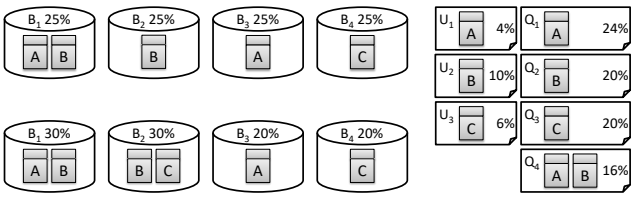


Figure 7: Optimal Update Aware Allocations on Homogeneous Backends (Above) and Heterogeneous Backends (Below)

The list *currentLoad* is updated to *currentLoad* $= (B_1 : 0.14, B_2 : 0, B_3 : 0, B_4 : 0)$. Since Q_4 is a read query class, as much of the remaining weight as possible is allocated to B_1. In this case, it is all weight of Q_4. *currentLoad* is updated, *currentLoad* $= (B_1 : 0.3, B_2 : 0, B_3 : 0, B_4 : 0)$ and the new load matrix is:

	Q_1	Q_2	Q_3	Q_4	U_1	U_2	U_3	Overall
B_1	0%	0%	0%	16%	4%	10%	0%	30%
B_2	0%	0%	0%	0%	0%	0%	0%	0%
B_3	0%	0%	0%	0%	0%	0%	0%	0%
B_4	0%	0%	0%	0%	0%	0%	0%	0%

The allocation matrix does not change. Since Q_4 is completely allocated, it is removed from $\mathfrak{C}$. Hence, *restWeight* does not need to be updated. At the end of the loop, $\mathfrak{C}$ is sorted again: $\mathfrak{C} = (Q_2, Q_1, Q_3)$. In the next execution of the loop, Q_2 is allocated. The differences to all backends are calculated: $(Q_2, B_1) : \infty, (Q_2, B_2) : 0, (Q_2, B_3) : 0, (Q_2, B_4) : 0$. Fragments referenced by Q_2 and U_2 are allocated to B_2. The resulting allocation matrix is:

	A	B	C
B_1	1	1	0
B_2	0	1	0
B_3	0	0	0
B_4	0	0	0

All load of the updates (U_2) is allocated to B_2, and since there still is enough load capacity for Q_2, it is also completely allocated to B_2. *currentLoad* is updated to *currentLoad* $= (B_1 : 0.3, B_2 : 0.3, B_3 : 0, B_4 : 0)$. The new load matrix is:

	Q_1	Q_2	Q_3	Q_4	U_1	U_2	U_3	Overall
B_1	0%	0%	0%	16%	4%	10%	0%	30%
B_2	0%	20%	0%	0%	0%	10%	0%	30%
B_3	0%	0%	0%	0%	0%	0%	0%	0%
B_4	0%	0%	0%	0%	0%	0%	0%	0%

Now, Q_2 is completely allocated and is removed from $\mathfrak{C}$. After sorting, $\mathfrak{C}$ becomes $\mathfrak{C} = (Q_1, Q_3)$. In the next iteration of the loop, Q_1 is allocated. The differences are: $(Q_1, B_1) : \infty, (Q_1, B_2) : \infty, (Q_1, B_3) : 0, (Q_1, B_4) : 0$. Therefore, all updates and as much weight as possible of Q_1 is allocated to B_3. The resulting allocation matrix is:

	A	B	C
B_1	1	1	0
B_2	0	1	0
B_3	1	0	0
B_4	0	0	0

Since the sum of the weights of Q_1 and its related update class U_1 is higher than *scaledLoad* of B_3, it is not completely allocated. *restWeight* is updated to *restWeight* $= (Q_1 : 0.08, Q_2 : 0.2, Q_3 : 0.2, Q_4 : 0.16)$. The *restWeight* of Q_2 and Q_4 were not updated, since they are not used anymore. The resulting load matrix is:

	Q_1	Q_2	Q_3	Q_4	U_1	U_2	U_3	Overall
B_1	0%	0%	0%	16%	4%	10%	0%	30%
B_2	0%	20%	0%	0%	0%	10%	0%	30%
B_3	16%	0%	0%	0%	4%	0%	0%	20%
B_4	0%	0%	0%	0%	0%	0%	0%	0%

Q_1 was not completely allocated and, hence, it is not removed from $\mathfrak{C}$. The result of the sorting is $\mathfrak{C} = (Q_3, Q_1)$. Q_3 is allocated to backend B_4. Again, it cannot be completely allocated. Lists *currentLoad* and *restWeight* are updated and the resulting allocation matrix is:

	A	B	C
B_1	1	1	0
B_2	0	1	0
B_3	1	0	0
B_4	0	0	1

The resulting load matrix is:

	Q_1	Q_2	Q_3	Q_4	U_1	U_2	U_3	Overall
B_1	0%	0%	0%	16%	4%	10%	0%	30%
B_2	0%	20%	0%	0%	0%	10%	0%	30%
B_3	16%	0%	0%	0%	4%	0%	0%	20%
B_4	0%	0%	14%	0%	0%	0%	6%	20%

Like Q_1, Q_3 was not allocated completely and therefore it remains in $\mathfrak{C}$. The result of sorting is: $\mathfrak{C} = (Q_1, Q_3)$. Since all backends are now at their maximum capacity, the relative capacity has to be scaled. This is done in relation to the original size of query class Q_1. Each backend is scaled so that it could hold a relative portion of the query class. For B_1 this is:

$$\text{scaledLoad}(B_1) = \text{currentLoad}(B_1) + \text{load}(B_1) \cdot \text{weight}(Q_1) = 0.372 \tag{34}$$

The updated *scaledLoad* is *scaledLoad* $= (B_1 : 0.372, B_2 : 0.372, B_3 : 0.248, B_4 : 0.248)$. The differences for Q_1 are: $(Q_1, B_1) : 0, (Q_1, B_2) : \mathit{size}(A), (Q_1, B_3) : 0, (Q_1, B_4) : \mathit{size}(A)$. Hence, as much weight of Q_1 as possible is allocated to B_1. Again, it cannot be completely allocated. The resulting *restWeight* is: *restWeight* $= (Q_1 : 0.008, Q_2 : 0.2, Q_3 : 0.06, Q_4 : 0.16)$. List *currentLoad* is updated to *currentLoad* $= (B_1 : 0.372, B_2 : 0.3, B_3 : 0.2, B_4 : 0.2)$. Q_2 stays in $\mathfrak{C}$. The allocation matrix has not changed; the new load matrix is:

	Q_1	Q_2	Q_3	Q_4	U_1	U_2	U_3	Overall
B_1	7.2%	16%	0%	0%	4%	10%	0%	37.2%
B_2	0%	20%	0%	16%	0%	10%	0%	30%
B_3	16%	0%	0%	0%	4%	0%	0%	20%
B_4	0%	0%	14%	0%	0%	0%	6%	20%

For Q_3, the differences are: $(Q_3, B_1) : \infty, (Q_3, B_2) : \mathit{size}(C), (Q_3, B_3) : \text{size}(C), (Q_3, B_4) : 0$. Therefore, Q_3 is allocated to B_4. The *restWeight* of Q_3 is 0.06. It cannot be allocated completely to B_4. The resulting *restWeight* and *currentLoad* are: *restWeight* $= (Q_1 : 0.008, Q_2 : 0.2, Q_3 : 0.012, Q_4 : 0.16)$ and *currentLoad* $= (B_1 : 0.372, B_2 : 0.3, B_3 : 0.2, B_4 : 0.248)$. The allocation matrix has not changed; the new load matrix is:

	Q_1	Q_2	Q_3	Q_4	U_1	U_2	U_3	Overall
B_1	7.2%	0%	0%	16%	4%	10%	0%	37.2%
B_2	0%	20%	0%	0%	0%	10%	0%	30%
B_3	16%	0%	0%	0%	4%	0%	0%	20%
B_4	0%	0%	18.8%	0%	0%	0%	6%	24.8%

After sorting, Q_1 is allocated. The differences are: $(Q_1, B_1) : \infty, (Q_1, B_2) : \text{size}(A), (Q_1, B_3) : 0, (Q_1, B_4) : \infty$. Therefore, it is completely allocated to backend B_3 and, thus, removed from $\mathfrak{C}$. The resulting load matrix is:

	Q_1	Q_2	Q_3	Q_4	U_1	U_2	U_3	Overall
B_1	7.2%	0%	0%	16%	4%	10%	0%	37.2%
B_2	0%	20%	0%	0%	0%	10%	0%	30%
B_3	16.8%	0%	0%	0%	4%	0%	0%	20.8%
B_4	0%	0%	18.8%	0%	0%	0%	6%	24.8%

Finally, only Q_3 remains. The differences are: $(Q_3, B_1) : \infty, (Q_3, B_2) : \text{size}(C), (Q_3, B_3) : \text{size}(C), (Q_3, B_4) : \infty$. It is allocated to backend B_2. First, the update class U_3 is allocated to B_2. Since the remaining capacity of B_2 is enough, Q_3 is completely allocated and the algorithm terminates. The resulting allocation and load matrices are:

	A	B	C
B_1	1	1	0
B_2	0	1	1
B_3	1	0	0
B_4	0	0	1

	Q_1	Q_2	Q_3	Q_4	U_1	U_2	U_3	Overall
B_1	7.2%	0%	0%	16%	4%	10%	0%	37.2%
B_2	0%	20%	1.2%	0%	0%	10%	6%	37.2%
B_3	16.8%	0%	0%	0%	4%	0%	0%	20.8%
B_4	0%	0%	18.8%	0%	0%	0%	6%	24.8%

B. OPTIMAL ALLOCATION

Based on the formal definition of the allocation presented in Section 3, a linear program is derived that computes an optimal allocation. The required constraints are represented as matrix. The allocation matrix $\mathfrak{A} \in \{0,1\}^{|\mathcal{B}|\times|F|}$ shows which data fragment is allocated to which backend.

$$\mathfrak{A}: \begin{cases} \{1,..,|\mathcal{B}|\}\times\{1,..,|F|\} \to \{0,1\} \\ (i,j) \mapsto a_{ij} \end{cases} \tag{35}$$

To express the load distributions of the query classes, two distribution matrices are used, $\mathfrak{L}_Q \in [0,1]^{|\mathcal{B}|\times|\mathcal{C}_Q|}$ and $\mathfrak{L}_U \in [0,1]^{|\mathcal{B}|\times|\mathcal{C}_U|}$.

$$\mathfrak{L}_Q: \begin{cases} \{1,..,|\mathcal{B}|\}\times\{1,..,|\mathcal{C}_Q|\} \to [0,1] \\ (i,k) \mapsto l_{ik} \end{cases} \tag{36}$$

$$\mathfrak{L}_U: \begin{cases} \{1,..,|\mathcal{B}|\}\times\{1,..,|\mathcal{C}_U|\} \to [0,1] \\ (i,k) \mapsto l'_{ik} \end{cases} \tag{37}$$

To ensure that all query classes are allocated, the Constraints 9 and 11 are satisfied. They are specified as follows:

$$\forall C_k \in \mathcal{C}_Q, l_{ik} \in \mathfrak{L}_Q : \sum_{i=1}^{|\mathcal{B}|} l_{ik} = \text{weight}(C_k) \tag{38}$$

$$\forall C_k \in \mathcal{C}_U, l'_{ik} \in \mathfrak{L}_U : \sum_{i=1}^{|\mathcal{B}|} l'_{ik} \geq \text{weight}(C_k) \tag{39}$$

Helper variables are needed to ensure that each update class is assigned to all backends, where a query class with the same data references is allocated. Two matrices of helper variables are defined $\mathfrak{H}_Q$ for read query classes and $\mathfrak{H}_U$ for update classes:

$$\mathfrak{H}_Q: \begin{cases} \{1,..,|\mathcal{B}|\}\times\{1,..,|\mathcal{C}_Q|\} \to \{0,1\} \\ (i,k) \mapsto h_{ik} \end{cases}, h_{ik} = \begin{cases} 1, \text{if } l_{ik} > 0 \\ 0, \text{else} \end{cases} \tag{40}$$

An update helper variable h'_{ik} must also be 1 if a read query class m is allocated to backend i that references the same data fragment as update query class k:

$$\mathfrak{H}_U: \begin{cases} \{1,..,|\mathcal{B}|\}\times\{1,..,|\mathcal{C}_U|\} \to \{0,1\} \\ (i,k) \mapsto h'_{ik} \end{cases}$$
$$h'_{ik} = \begin{cases} 1, \text{if } l'_{ik} > 0 \\ 1, \text{if } \exists l_{im} \in \mathfrak{L}_Q : l_{im} > 0 \text{ and } C_k \in \text{updates}(C_m) \\ 0, \text{else} \end{cases} \tag{41}$$

Using the helper variables, Constraint 10 is specified as follows:

$$\forall C_k \in \mathcal{C}_U, l'_{ik} \in \mathfrak{L}_U : l'_{ik} = h'_{ik} * \text{weight}(C_k) \tag{42}$$

The load constraints of the backends must be considered. According to Equation 15, this can be expressed using the factor $scale$:

$$\forall B_i \in \mathcal{B}, l_{ik} \in \mathfrak{L}_Q, l'_{ik} \in \mathfrak{L}_U : \sum_{k=1}^{|\mathcal{C}_Q|} l_{ik} + \sum_{m=1}^{|\mathcal{C}_U|} l'_{im} \leq scale * \text{load}(B_i) \tag{43}$$

Finally, all fragments which are referenced by a read query class have to be allocated to the backends where the query class is allocated:

$$\forall C_k \in \mathcal{C}_Q, \forall B_i \in \mathcal{B} : \sum_{j:f_j \in C_k} a_{ij} \geq |C_k| * h_{ik} \tag{44}$$

The same must be true for the fragments referenced by update query classes:

$$\forall C_k \in \mathcal{C}_U, \forall B_i \in \mathcal{B} : \sum_{j:f_j \in C_k} a_{ij} \geq |C_k| * h'_{ik} \tag{45}$$

A throughput optimal allocation can be found by minimizing the factor $scale$. The definitions above can be directly used by a solver. To find an allocation with minimal space requirements, a second linear program is formulated including the previously computed optimal $scale$ that minimizes the total allocated space.

C. K-SAFETY

In distributed systems, error rates are multiplied by the number of components. Therefore, in large scale clusters, failures are a daily business. For example, Google published numbers that revealed each of their clusters of 1800 nodes had over 1000 machine failures and thousands of hard disk failures in the first year [54]. Hence, distributed systems have to include mechanisms to deal with hardware failures.

On the cluster level, the standard fault tolerance approach is redundancy [28, 49]. Therefore, we present an extension to our algorithm that introduces k-safety [56]. With k-safety, the algorithm ensures that the loss of k backends can be tolerated. If each fragment is allocated to at least $k+1$ backends, no data is lost if k backends fail. However, to ensure that all queries can still be processed locally without a reallocation, each query class has to be allocated to at least $k+1$ backends. The basis is still the CDBS processing model introduced in Section 2; hence, updates can only be processed on backends that have all referenced data. As a result, the independent allocation of fragments that are updated is not possible. According to equation 10, update query classes have to be allocated completely to each backend.

In the read-only case, the introduction of k-safety has the drawback of increased space requirements. The theoretical speedup is unaffected by the additional replicas. To the contrary, additional replicas allow for a more flexible load balancing, especially, if the load is slightly varying. In practice, however, the additional replicas will result in a less fragmented schema and, hence, larger relations. In some cases, especially, for large relations, this will reduce the performance due to an increased cache miss rate. For the update sensitive case, replication reduces performance, if the replicas introduce replicated updates. The formal definitions and algorithms presented before can easily be adapted. First, we explain the extensions for the allocation of $k+1$ replicas of each fragment for the read-only case and then the allocation of each query class to $k+1$ backends for both

cases. Since our processing model enforces local execution of query classes, k-safety of data fragments is not enough to ensure that a system is still capable of processing all queries. Hence, we present a second definition of k-safety which ensures that a CDBS can tolerate the loss of k backends and still process all incoming queries.

C.1 Redundant Fragments

The independent replication of fragments without replicating complete query classes is in general only possible for read-only access of data. In the presence of updates, the read-only fragments can still be replicated without considering the query classes. We do not elaborate on this option, since the approach is a straightforward combination of fragment and query class replication. If $k+1$ copies of each fragment have to be allocated, the following constraint are introduced to the formal definition in Section 3.2:

$$\forall f \in F: \sum_{\{B \in \mathcal{B} \mid f \in \text{fragments}(B)\}} 1 \geq k+1 \tag{46}$$

The equation ensures that each fragment is allocated at least $k+1$ times. Without further restrictions, a straightforward solution in the read-only case is to place fragments that have to be further allocated to the first backends to which they are not already allocated. To ensure a better distribution of the additional data, a randomized approach will distribute fragments more evenly. In the update sensitive case, only fragments that are never updated are freely placed. The placement of fragments with updates is part of the optimization goal. We introduce an extended algorithm for this case below.

C.2 Redundant Query Classes

To ensure that the CDBS is fully operational after the loss of k backends, each query class has to be allocated to at least $k+1$ backends. In the formal definition (see Section 3.2), the following constraint ensures k-safety:

$$\forall C \in \mathcal{C}: \sum_{B \in \mathcal{B}} \begin{cases} 1, \text{if assign}(C,B) > 0 \\ 0, \text{else} \end{cases} \geq k+1 \tag{47}$$

Since replicated updates increase the workload and, thus, potentially reduce the overall throughput, the replicated query classes are treated like other query classes and are allocated accordingly. This can be done by introducing a new set $\mathcal{C}_k$, which contains the query classes that have to be further replicated. Initially, $\mathcal{C}_k$ is empty. Whenever a query class $C \in \mathcal{C}_Q$ is completely allocated, the number of backends that it has been allocated to, is counted and stored in $replicas$ and if $replicas < k+1$, then $k+1-replicas$ copies of C are added to $\mathfrak{C}$ and $\mathcal{C}_k = \mathcal{C}_k \cup \{C\}$. This is done by the pseudo-code given in Algorithm 3.

```
35 𝔅 ← {B ∈ ℬ | C \ fragments(B) = ∅};
36 replicas ← |𝔅|;
37 if replicas < k+1 then
38 |   𝔠 ← 𝔠 ∪ ((k+1−replicas)·C);
39 |   C_k = C_k ∪ {C};
```

Algorithm 3: Adding Missing Replicas of Query Classes

These replicated queries all have no weight, except for the update classes that have to be additionally allocated. Hence, they can be treated like update classes and be allocated each to a single backend. If the weight of the additional updates overload the backend, the *scaledLoad* has to be adapted, hence, the condition of the *if* statement in Line 20 is altered to $C \in \mathcal{C}_U \vee C \in \mathcal{C}_k$. Finally, it has to be assured that replicated query classes are not allocated to backends, which already contain a replica. This can be done by setting the difference to ∞. To ensure that all update classes are allocated $k+1$ times, those that are not allocated with query classes have to be added $k+1$ times to $\mathfrak{C}$. The complete pseudo-code is given in Algorithm 4.

Input: Classification $\mathcal{C}$, set of empty backends $\mathcal{B}$, degree of redundancy $k+1$
Output: Heuristic allocation $\mathcal{B}$

```
1  C* ← C_Q ∪ {C_U ∈ C_U | ∄ C_Q ∈ C_Q : C_U ∩ C_Q ≠ ∅};
2  C_k ← {C_U ∈ C_U | ∄ C_Q ∈ C_Q : C_U ∩ C_Q ≠ ∅};
3  𝔠 ← sort C ∈ C* descending to
     weight(C ∪ updates(C)) × size(C ∪ updates(C));
4  currentLoad(ℬ) ← 0;
5  scaledLoad(ℬ) ← load(ℬ);
6  restWeight(𝔠) ← weight(𝔠);
7  while C ∈ 𝔠 do
8  |   if all backends are full then
9  |   |   foreach B ∈ ℬ do
10 |   |   |   scaledLoad(B) ← currentLoad(B) + load(B)·weight(C);
11 |   foreach B ∈ ℬ do
12 |   |   if (currentLoad(B) = scaledLoad(B)) ∨ ((assign(B,C) >
   |   |     0) ∧ C ∈ C_k) then
13 |   |   |   difference(C,B) ← ∞;
14 |   |   else if currentLoad(B) = 0 then
15 |   |   |   difference(C,B) ← 0;
16 |   |   else
17 |   |   |   difference(C,B) ← size((C ∪ updates(C)) \ fragments(B));
18 |   B ← B ∈ ℬ with difference(C,B) minimal;
19 |   fragments(B) ← fragments(B) ∪ C ∪ updates(C);
20 |   currentLoad(B) ←
   |     currentLoad(B) + weight(updates(C)) − updateWeight(B,C);
21 |   if (C ∈ C_U) ∨ (C ∈ C_k) then
22 |   |   if currentLoad(B) > scaledLoad(B) then
23 |   |   |   scaledLoad(B) ← currentLoad(B);
24 |   |   𝔠 ← 𝔠 \ {C};
25 |   else
26 |   |   if currentLoad(B) ≥ scaledLoad(B) then
27 |   |   |   scaledLoad(B) ← currentLoad(B) + load(B)·weight(C);
28 |   |   if restWeight(C) > scaledLoad(B) − currentLoad(B) then
29 |   |   |   restWeight(C) ←
   |   |   |     restWeight(C) − (scaledLoad(B) − currentLoad(B));
30 |   |   |   currentLoad(B) ← scaledLoad(B);
31 |   |   else
32 |   |   |   currentLoad(B) ← currentLoad(B) + restWeight(C);
33 |   |   |   𝔠 ← 𝔠 \ {C};
34 |   |   |   𝔅 ← {B ∈ ℬ | C \ fragments(B) = ∅};
35 |   |   |   replicas ← |𝔅|;
36 |   |   |   if replicas < k+1 then
37 |   |   |   |   𝔠 ← 𝔠 ∪ ((k+1−replicas)·C);
38 |   |   |   |   C_k = C_k ∪ {C};
39 |   sort(𝔠) descending to restWeight and size;
40 return ℬ
```

Algorithm 4: Greedy Allocation Algorithm with K-Safety

The algorithm for k-safety works similar to the original algorithm. The first difference is the generation of the multi-set $\mathcal{C}_k$ in Line 2; it is the set of update classes that need to be explicitly allocated k additional times. While update classes and replicated query classes are allocated as before, we additionally add as many replicas of the read query classes as are still required to $\mathfrak{C}$ after the first full allocation. This is done as shown in the excerpt in Algorithm 3. Apart from the greedy heuristic, also, the meta heuristic has to be adapted, due to space limitations, we do not discuss this extension here. Replicating every data fragment multiple times reduces the probability of a data loss enormously.