Tim Kiefer, Dirk Habich, Wolfgang Lehner

Penalized graph partitioning based allocation strategy for database-as-a-service systems

Erstveröffentlichung in / First published in:

UCC '16: 9th International Conference on Utility and Cloud Computing, Shanghai 06.12. – 09.12.2016. ACM Digital Library, S. 200–209. ISBN 978-1-4503-4617-7

DOI: https://doi.org/10.1145/3006299.3006300

Penalized Graph Partitioning based Allocation Strategy for Database-as-a-Service Systems

Tim Kiefer, Dirk Habich, Wolfgang Lehner
Technische Universität Dresden
Database Systems Group
Dresden,Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

Databases as a service (DBaaS) transfer the advantages of cloud computing to data management systems, which is important for the big data era. The allocation in a DBaaS system, i.e., the mapping from databases to nodes of the infrastructure, influences performance, utilization, and cost-effectiveness of the system. Modeling databases and the underlying infrastructure as weighted graphs and using graph partitioning and mapping algorithms yields an allocation strategy. However, graph partitioning assumes that individual vertex weights add up (linearly) to partition weights. In reality, performance does usually not scale linearly with the amount of work due to contention on the hardware, on operating system resources, or on DBMS components. To overcome this issue, we propose an allocation strategy based on penalized graph partitioning in this paper. We show how existing algorithms can be modified for graphs with non-linear partition weights, i.e., vertex weights that do not sum up linearly to partition weights. We experimentally evaluate our allocation strategy in a DBaaS system with 1,000 databases on 32 nodes.

Keywords

Load Balancing, Database-as-a-Service, Query Processing, Allocation

1. INTRODUCTION

In the era of big data, cloud computing and big data are conjoined, because cloud services have become a powerful architecture to perform complex large-scale tasks on large data sets [8, 12]. Offering (relational) databases as a cloud service (DBaaS) transfers the advantages of cloud computing to data management systems [8]. All major cloud service providers like Amazon, Microsoft, or Oracle offer relational databases as part of their cloud product portfolio. As shown in [8], database partitioning plays an important role for DBaaS. On the one hand, data partitioning enables to scale a single database to multiple back-end nodes of the cloud infrastructure which is useful when the load exceeds the capacity of a single machine [8]. On the other hand, it offers more fine-grained load balancing opportunities on the back-end nodes compared to placing entire databases [8].

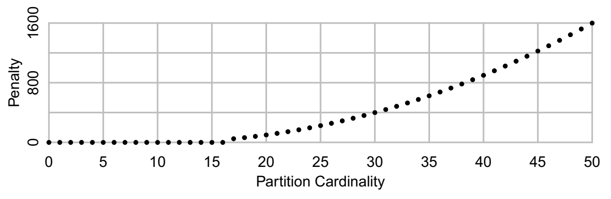
In this context, the *allocation* in a DBaaS system refers to the mapping from database (or database partitions) to the back-end nodes (i.e., servers) of the underlying infrastructure. Bad allocations cause skew in the load and therefore suboptimal application performance and infrastructure utilization. Even on a small scale, solving the allocation problem is hard, given the complex interactions that tasks may have [1]. Previous approaches to the allocation problem, which are often based on variations of bin-packing or integer programming [7, 21, 23], concentrate on balancing the load and thereby minimizing the cost caused by the number of machines or penalties defined in service-level agreements (SLAs). None of these approaches takes the communication into account. However, communication in DBaaS systems, in particular for answering complex analytical queries, is a dominant factor due to the data partitioning which is required for scalability[8].

To tackle the communication aspect, the databases in a DBaaS system can be modeled as a weighted graph (workload graph), where vertex weights represent resource requirements and edge weights represent communication costs between (partitions of) databases. The underlying infrastructure of the DBaaS can also be modeled as a weighted graph (infrastructure graph), i.e., back-end servers with resource capacities connected by a network. Using these abstractions, graph partitioning and mapping algorithms are perfectly suited to solve the allocation problem in DBaaS systems. Graph partitioning, more specifically balanced k-way min-cut partitioning, is successfully used in other contexts to optimize the allocation in, e.g., OLTP systems [9, 26]. The goal of the balanced k-way min-cut problem is to partition a graph into k parts such that the sum of edges that is cut is minimized while keeping the sizes of all parts balanced. Applied to our DBaaS workload graph, a partitioning minimizes communication and at the same time balances load across the nodes.

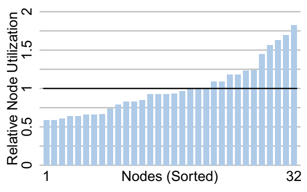
However, there is a mismatch between classic graph partitioning and many real back-end nodes. Graph partitioning usually assumes that individual vertex weights add up to partition weights (here, referred to as *linear graph partitioning*). In the context of DBaaS systems, this implies that the load induced by all databases (or partitions) that share a node equals the sum of all individual loads and that therefore performance scales linearly with the number of databases.

DOI: http://dx.doi.org/10.1145/3006299.3006300

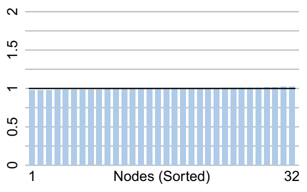
Final edited form was published in "UCC '16: 9th International Conference on Utility and Cloud Computing. Shanghai 2016", S. 200–209, ISBN 978-1-4503-4617-7
https://doi.org/10.1145/3006299.3006300



(a) Exponential Penalty Function



(b) Skewed Node Utilization



(c) Penalized Node Utilization

Figure 1: Partitioning Experiment (Node Utilization Normalized to the Average Node Utilization)

In reality, performance does usually not scale linearly with the amount of work due to contention on hardware [4], operating system [22], or application resources [25]. We address this mismatch with our novel *penalized graph partitioning* approach which offers the advantages of classic graph partitioning and considers the non-linear performance behavior of real back-end nodes at the same time.

1.1 Motivating Example

To demonstrate the potential of our penalized graph partitioning in presence of non-linear resources, we performed a synthetic partitioning experiment. We generated a workload graph that contains 1,000 heterogeneous tasks with weights following a Zipf distribution as found in typical DBaaS systems [27]. Each task in the workload graph is communicating with 0 to 10 other tasks (again Zipf distributed). To model a system, we use an exponential penalty function and assume that the underlying resource can execute 16 parallel tasks before the penalty grows with the square of the cardinality due to contention (Figure 1a).

The workload in this experiment is partitioned into 32 balanced partitions using a standard graph partitioning library. Afterwards, to estimate the actual load for each node, the penalty function is applied to each partition based on the partition cardinality (Figure 1b). The resulting partition weights are compared to a second partitioning of the graph that was generated by our novel penalized graph partitioning algorithm (Figure 1c).

The unmodified partitioning algorithm, which is unaware of the contention, tries to balance the load. The resulting relative weights show that the node with the highest partition weight receives 3.1 times the load of the node with the lowest partition weight. In contrast, the penalized partitioning algorithm leads to partition weights, and hence node utilizations, that are balanced within a tolerance of 3%.

1.2 Contributions and Outline

Our main contribution in this paper is a novel allocation strategy for DBaaS systems that considers communication between databases as well as non-linear performance of the underlying system. In detail, we present: (1) an introduction to graph partitioning (Section 2), (2) a model for DBaaS systems, specifically the infrastructure and the workload, using weighted graphs (Section 3), (3) a method to partition graphs with non-linear partition weights as foundation for our allocation strategy (Section 4), (4) an experimental evaluation of the scalability of our penalized graph partitioning in Section 5, (5) an evaluation of our allocation strategy in a DBaaS System with 32 nodes and 1,000 databases using an in-memory DBMS and OLAP workloads (Section 6). Finally, we conclude the paper with related work and a summary in Sections 7 and 8.

2. GRAPH PARTITIONING

Given an undirected, weighted graph, the balanced k-way min-cut graph partitioning problem (GPP) refers to finding a k-way partitioning of the graph such that the total edge cut is minimized and the partitions are balanced within a given tolerance [2]. Here, we limit ourselves to graphs with a single weight per vertex. Balancing graphs with multiple vertex weights is also well-known and researched in the literature as the *multi-constraint graph partitioning problem* (MC-GPP) [16]. Without restriction, our methods for penalized graph partitioning work with multiple vertex weights as well (e.g., based on [16]).

Let $G = (V, E, w_V, w_E)$ be an *undirected, weighted graph* with vertices V, edges E, and weight functions w_V and w_E. Vertex and edge weights are positive real numbers: $w_V : V \to \mathbb{R}_{>0}$ and $w_E : E \to \mathbb{R}_{>0}$. The weight functions are extended to sets of vertices and edges:

$$w_V(V') := \sum_{v \in V'} w_V(v) \text{ for } V' \subseteq V \text{ and}$$

$$w_E(E') := \sum_{e \in E'} w_E(e) \text{ for } E' \subseteq E.$$

Let $\Pi = (V_1, \ldots, V_k)$ be a *partitioning* of V into k partitions $V_1, \ldots, V_k$ such that: $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for all $i \neq j$. Given a partitioning, an edge that connects partitions is called a *cut edge*. E_c is the set of all cut edges in a graph. The objective of the GPP is to minimize the *total cut* $w_E(E_c)$, i.e., the aggregated weight of all cut edges. Furthermore, a *balance constraint* demands that all partitions have about equal weights. Let $\mu := w_V(V)/k$ be the average partition weight. For a partitioning to be balanced it must hold that $\forall i \in \{1, \ldots, k\}$: $w_V(V_i) \leq (1 + \epsilon) \cdot \mu$, where $\epsilon \in \mathbb{R}_{\geq 0}$ is an imbalance parameter determined by the application.

Given a partitioning, the *internal degree* $id(v)$ of a vertex $v \in V_i$ is the accumulated edge weight of all edges that connect v to vertices in the same partition. The *external degree* $ed_j(v)$ of a vertex $v \in V_i$ with respect to partition V_j is the accumulated edge weight of all edges that connect v to vertices in partition V_j. The *gain* of a vertex $v \in V_i$ with respect to partition V_j is the reduction in cut when v is moved from V_i to V_j, i.e., $g_j(v) := ed_j(v) - id(v)$. Note that the gain may be negative, which implies that moving the vertex increases the cut.

Partitioning Algorithms

Partitioning a graph into k partitions of roughly equal size such that the total cut is minimized is NP-complete [14]. Heuristics, especially the multilevel partitioning framework, are used in practice to solve the graph partitioning problem [15]. This framework consists of three phases: (1) *coarsening* the graph, (2) finding an *initial partitioning* of the coarse graph, and (3) *uncoarsening* the graph and projecting the coarse solution to the finer graphs.

In the **coarsening phase**, a series of smaller graphs is derived from the input graph. Coarsening is implemented by contracting a subset of vertices and replacing it with a single vertex. Parallel edges are replaced by a single edge with the accumulated weight of the parallel edges. Contracting vertices like this implies that a balanced partitioning on the coarse level represents a balanced partitioning on the fine level with the same total cut. Different strategies exist to select vertices to be contracted. Finding a matching is a tradeoff between using heavy edges (and hence reducing the final cut) and keeping uniform vertex weights (and hence improving partition balance). The coarsening ends when the coarsest graph is sufficiently small to be initially partitioned.

Different algorithms exist to find an **initial partitioning** [5]. Methods for the initial partitioning are either based on direct k-way partitioning or on recursive bisection. A simple but effective method to find an initial partitioning is greedy graph growing. A random start vertex is grown using breadth-first search, adding the vertex that increases the total cut the least in each step. The search is stopped as soon as half of the total vertex weight is assigned to the growing partition. Because the quality of the bisection strongly depends on the randomly selected start vertex, multiple iterations with different starts are used and the best solution is kept. The k-way extension of graph-growing starts with k random vertices and grows them in turns.

The initial partitioning is **uncoarsened** by repeatedly assigning previously contracted vertices to the same partition. Each extraction of vertices is followed by a **refinement step** to improve the total cut or the balance of the partitions. Local vertex swapping is a refinement metaheuristic that can be parametrized with different strategies to select vertices to move. In [17], the authors propose to move the vertex with the highest value of the gain function, i.e., the vertex that yields the largest decrease in total cut. Each vertex is considered only once per round and rounds are repeated until there is no further improvement. The method is known as Kernighan-Lin (KL) method. KL/FM[11] improved the KL method with carefully designed data structures.

3. DBAAS ALLOCATION PROBLEM

In this section, we formalize the infrastructure, the workload, and the allocation problem for DBaaS systems.

3.1 Infrastructure Model

The *infrastructure* of a DBaaS system is an undirected graph of *nodes* connected by *links* as shown in Figure 2. Nodes have bounded and unbounded resources and weight functions map each node to either an absolute or a relative capacity per resource. Each unbounded resource additionally has a model for combining loads. Link capacities in the infrastructure are given by an edge-weight function.

Generally, different kinds of hardware node resources can be modeled with our infrastructure approach, e.g., processing resources (CPU cycles), memory resources (bandwidth, capacity), and network resources (link bandwidth). Furthermore, our infrastructure model distinguishes between *bounded resources* and *unbounded resources*. Bounded resources have a hard limit that cannot be exceeded (e.g., memory capacity). Overloading a node's bounded resources leads to an invalid allocation. Unbounded resources are not

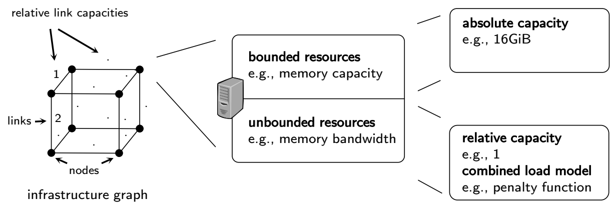


Figure 2: Infrastructure Components.

literally unbounded but they can (and usually will) be overcommitted (e.g., memory bandwidth). Overcommitting an unbounded resource still leads to a valid allocation, even if the performance degrades. Modeling resources as being bounded or unbounded depends on the intention of the model and the availability of resource capacity and workload requirement information. The goal of our allocation strategy will be to respect the upper bounds of bounded resources and to balance unbounded resources.

Several databases (or partitions) share a single server in DBaaS systems. To be able to evaluate a given allocation, the aggregated load of a node (induced by all tasks) needs to be estimated. In the simplest case, loads induced by tasks are combined by summing them up to derive a node's global load. This method is referred to as the *linear model* as it models an ideal system where performance scales linearly with the amount of work that needs to be done. However, performance does usually not scale linearly with the amount of work due to contention on hardware [3], operating system [22], or application resources [25]. Therefore, in addition to the linear model, we use a *non-linear* model to combine the individual loads induced by tasks that share a node.

To grasp the behavior of the complex systems, we assume a simplified penalized resource consumption model reflecting the non-linear performance of the system. Our *penalized resource usage model*, which we believe to be applicable in many real systems, is a combination of the linear model and a penalty function. Up to a certain load or degree of parallelism, the linear usage assumption often holds. However, contention occurs and the performance does not scale linearly when a certain load level is reached. To account for this contention and the non-linear performance, a penalty is introduced that adds to the resource usage based on the number of concurrent tasks.

While we acknowledge that modeling real systems is a challenging problem in itself, we assume here that the model, i.e., the penalty function, is given. Depending on the infrastructure, low-level and application-level experiments (like in [21, 23]) may be necessary to find a sufficiently accurate model.

3.2 Workload Model

The *workload* in a DBaaS system is also an undirected graph of *database partitions* connected by *data transfers*. Database partitions and operations executed on them are bundled in *tasks*. Tasks consume bounded and unbounded resources and vertex weight functions map each task to either an absolute or a relative load per resource. An edge weight function quantifies data transfer costs. The workload graph can, e.g., be derived from a list of SQL queries by using the query execution plans and cost estimates from the DBMS. Like any workload-driven approach, we assume that the workload of the system is known or can be captured and

Final edited form was published in "UCC '16: 9th International Conference on Utility and Cloud Computing. Shanghai 2016", S. 200–209, ISBN 978-1-4503-4617-7
https://doi.org/10.1145/3006299.3006300

monitored. This assumption seems reasonable, especially in DBaaS systems where workload characteristics are captured for billing purposes anyway. Note that we assume an existing partitioning of databases, i.e., the granularity of tasks in the workload [8]. Our focus in this paper is the allocation of these existing partitions to infrastructure nodes.

3.3 Allocation Problem Formulation

An *allocation* is a mapping from workload tasks to infrastructure nodes. The allocation problem can be formulated differently to optimize them for different objectives. Two basic formulations are (1) to minimize the number of used resources without violating performance constraints or (2) to maximize performance with a given amount of resources. Our allocation strategy is based on the latter formulation. To solve this allocation objective, two steps are necessary. In the first step, the workload graph is partitioned using a graph partitioning algorithm. The presented balanced k-way min-cut algorithms can be applied to partition a given graph into k parts (k nodes are available in the infrastructure) such that the sum of edges that are cut is minimized while keeping the sizes of all parts within balance. Then, the mapping of the determined partitions to infrastructure nodes is conducted in the second step.

However, there is a mismatch between the classical graph partitioning approach (GPP or MC-GPP) and the actual behavior of the infrastructure. A fundamental assumption in the graph partitioning problem is that vertex weights (individually) sum up to reflect the weight of a partition. The implication is that, no matter how many vertices comprise a partition, the weight of the partition is the sum of all participating vertices. Translated to the infrastructure, this means that no matter how many tasks are executed on a node, the combined load of the node is the sum of all loads caused by the tasks. As mentioned in our infrastructure model, performance on actual hardware resources usually does not scale linearly with the number of concurrent tasks. Taking this aspect into account for the allocation in DBaaS systems, we reformulate the graph partitioning problem as follows:

An *allocation* is a mapping from workload tasks to infrastructure nodes. For each resource, a node's load is the combined weight (including non-linear behavior) of all tasks assigned to that node. An allocation is *valid* if no load is greater than the node's capacity for all bounded resources. An allocation is *balanced* if all loads are equal (within a tolerance) to the average load for all unbounded resources. An allocation's *communication costs* are the sum of all edge weights of edges between vertices that are assigned to different nodes.

4. ALLOCATION STRATEGY BASED ON PENALIZED GRAPH PARTITIONING

To solve the DBaaS allocation challenge, we present our novel penalized graph partitioning approach. To improve the readability of this paper, we limit ourselves to graphs with a single weight per vertex—assuming a single (unbounded) resource. Nevertheless, our presented methods also work with multiple vertex weights without restrictions (e.g., based on [16]). Furthermore, we assume a static workload and a homogeneous infrastructure. Later in this section, we present extensions of the basic methods to relax some of these restrictions.

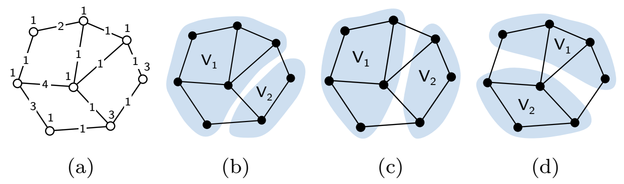


Figure 3: Example of Graph Partitionings with Different Penalty Functions; (a) V/E Weights, (b) No Penalty, (c) Linear Penalty, (d) Square Penalty

4.1 Penalized Graph Partitioning

The idea of our penalized graph partitioning is to introduce a *penalized partition weight* with respect to our *penalized resource usage model* and to modify the graph partitioning problem accordingly. We define the resulting problem as the *Penalized Graph Partitioning Problem* (P-GPP). Figure 3 shows an example graph with vertex and edge weights denoted in Figure 3a. Solving the GPP leads to the partitioning with the total cut of 3 shown in Figure 3b. When the cardinality of a partition is penalized linearly, the solution of the P-GPP having a total cut of 4 is shown in Figure 3c. However, when the penalty of a partition grows with the square of the partition cardinality, the partitioning with the total cut of 4 shown in Figure 3d is the solution to the P-GPP. The partitioning obviously depends on the performance model, i.e., the given penalty function.

4.1.1 Prerequisites

Let $G = (V, E, w_V, w_E)$ be an undirected, weighted graph as in Section 2. Furthermore, let p be a positive, monotonically increasing penalty function that penalizes a partition weight based on the partition cardinality:

$$p\colon \mathbb{N} \to \mathbb{R}_{\geq 0} \text{ with } p(n_1) \leq p(n_2) \text{ for } n_1 \leq n_2.$$

The vertex weight function is extended to sets $V' \subseteq V$ such that it incorporates the penalty:

$$w_V(V') := \sum_{v \in V'} w_V(v) + p(|V'|).$$

The example partitioning in Figure 3c uses a linear penalty function, i.e., $p(|V|) := |V|$. Accordingly, using the definition, the partition weights are

$$w_V(V_1) = \sum_{v \in V_1} w_V(v) + p(|V_1|) = 5 + 5 = 10 \text{ and}$$

$$w_V(V_2) = \sum_{v \in V_2} w_V(v) + p(|V_2|) = 7 + 3 = 10.$$

The example partitioning in Figure 3d uses a square penalty function, i.e., $p(|V|) := |V|^2$. Accordingly, the partition weights are $w_V(V_1) = w_V(V_2) = 22$.

Adding penalties to partition weights invalidates some of the assumptions made in the GPP and its solution algorithms. Most fundamentally, the combined weight of two or more partitions is not equal to the weight of a partition containing all the vertices. Using the definition and two partitions V_1 and V_2:

$$\begin{aligned} w_V(V_1 \cup V_2) &= \sum_{v \in V_1 \cup V_2} w_V(v) + p(|V_1 \cup V_2|) \\ &= w_V(V_1) + w_V(V_2) + p(|V_1 \cup V_2|) - p(|V_1|) - p(|V_2|). \end{aligned}$$

For arbitrary penalty functions we must assume that $p(|V_1 \cup V_2|) \neq p(|V_1|) + p(|V_2|)$. It follows that in general $w_V(V_1 \cup V_2) \neq w_V(V_1) + w_V(V_2)$. Hence, the total weight of all vertices is in general not equal to the total weight of all partitions. We therefore introduce the following definitions of the two weights. Given a graph and a partitioning, the *total vertex weight* w_V is the penalized weight of all vertices, i.e.,

$$w_V := \sum_{v \in V} w_V(v) + p(|V|).$$

The *total partition weight* w_Π on the other hand is the sum of the weights of all partitions, i.e.,

$$w_\Pi := \sum_{i=1}^{k} w_V(V_i).$$

Consider the example partitioning in Figure 3d; using the definition, $w_V = 12 + 64 = 76$ and $w_\Pi = 22 + 22 = 44$.

It follows that the total partition weight w_Π of the graph is not constant but depends on the partitioning, specifically the cardinalities of the partitions. This observation has implications in all steps of the graph partitioning algorithm, e.g., the balance constraint has to use the average total partition weight $\mu := w_\Pi/k$ instead of the average total vertex weight.

4.1.2 *Algorithm (Static Case)*

We propose modifications of the multilevel graph partitioning algorithm to solve the P-GPP. First, we describe two basic operations that need to reflect partition penalties. Then, we will detail the necessary modifications to the three building blocks of the multilevel graph partitioning framework.

During graph partitioning and refinement, it is often necessary to move a vertex between partitions or to merge partitions. For the sake of computational efficiency, the weights of the resulting partitions should be computed incrementally instead of from scratch.

OPERATION 1. *When a vertex v is moved from partition V_1 to partition V_2, the partition weights of the resulting partitions $V_1' := V_1 \setminus v$ and $V_2' := V_2 \cup v$ are as follows:*

$$\begin{aligned} w_V(V_1') &= w_V(V_1) - w_V(v) - p(|V_1|) + p(|V_1| - 1), \\ w_V(V_2') &= w_V(V_2) + w_V(v) - p(|V_2|) + p(|V_2| + 1). \end{aligned}$$

OPERATION 2. *When two partitions V_1 and V_2 are combined, the partition weight of the resulting partition $V' := V_1 \cup V_2$ can be calculated as follows:*

$$\begin{aligned} w_V(V') = w_V(V_1 \cup V_2) &= \sum_{v \in V_1 \cup V_2} w_V(v) + p(|V_1 \cup V_2|) \\ &= w_V(V_1) + w_V(V_2) + p(|V_1| + |V_2|) - p(|V_1|) - p(|V_2|). \end{aligned}$$

To **coarsen** the graph, a matching of vertices has to be determined and vertices have to be contracted accordingly. The heuristics introduced in Section 2 can be used to coarsen a graph with penalized partition weights. However, the vertex weight of the contracted vertex has to correctly incorporate the penalty to ensure that a balanced partitioning of the coarse graph will lead to a balanced partitioning during the uncoarsening steps. Therefore, contracted vertices are treated like partitions themselves and the weight of a contracted vertex is calculated as in Operation 2.

We use a modified version of recursive bisection and greedy region growing to find an **initial k-way partitioning** of graphs with penalized partition weights. In the region growing algorithm, moving a vertex between partitions has to use Operation 1 to calculate the resulting partition weights. Moreover, the stop condition of the region growing algorithm has to be modified to account for the new balance constraint. In the original formulation, the algorithm stopped when the growing partition reached at least half of the total vertex weight. To achieve balanced partitions and because the total vertex weight is in general not equal to the total partition weight, the latter has to be used in the stop condition. Furthermore, since the total partition weight depends on the partitioning it repeatedly has to be recalculated after vertices have been moved, again using Operation 1.

The penalties have to be considered during the **uncoarsening and refinement** of the graph. Similar to the modifications of the region growing algorithm, the local vertex swapping method has to use Operation 1 whenever a vertex is moved between partitions. Furthermore, when vertex swapping is used to balance a partitioning, the modified balance constraint has to be used. This implies that stop conditions and checks use the total partition weight instead of the total vertex weight. Since the total partition weight depends on the partitioning, it has to be recalculated after a vertex has been moved (Operation 1).

4.1.3 *Incrementally Updating the Partitioning (Dynamic Case)*

With dynamic DBaaS workloads, the partitioning needs to be periodically re-evaluated to ensure balanced partitions and an optimal total cut. Updating the partitioning after changes is a tradeoff between the quality of the new partitioning and the migration costs induced by implementing the new partitioning.

The problem of incrementally updating a partitioning is known as dynamic load balancing or repartitioning and is a well studied problem for the original graph partitioning problem [6, 10]. We are able to adapt existing hybrid update strategies for our penalized graph partitioning. Whenever the graph changes such that the balance constraint is violated, balancing and refinement steps based on local vertex swapping try to move vertices such that the partitioning is balanced again. If no balanced partitioning can be found using the local search strategy, the graph is partitioned from scratch and the new partitioning is mapped to the previous partitioning such that the migration cost is minimized. To prevent the total cut in the graph from slowly deteriorating, a new partitioning is computed in the background after a certain number of local refinement operations (even when the partitioning is still balanced). The new partitioning replaces the current one only if the new total cut justifies the migration overhead.

4.2 DBaaS Allocation Strategy

The presented penalized graph partitioning approach is the foundation of our DBaaS allocation strategy. Applied to the workload graph, we are able to determine balanced partitions when the performance of the back-end nodes does not scale linearly with the amount of work. The non-linear behavior is modeled as a penalty function. Furthermore, the communication is also considered in the partitioning. Then, the determined workload partitions are mapped to the infrastructure nodes.

4.3 Extensions to Graph Partitioning

Moreover, we propose two extensions to our graph partitioning to relax previously made assumptions.

Heterogeneous Infrastructures. In the presence of a heterogeneous infrastructure, finding an optimal mapping from partitions to infrastructure nodes becomes part of the GPP. Let $I = (N, L, w_N, w_L)$ be the undirected, weighted infrastructure graph of nodes (N) connected by links (L).

We first consider an infrastructure with **heterogeneous nodes**, but a homogeneous communication network. The balance constraint of the GPP can be modified to account for different node capacities. The *heterogeneous balance constraint* demands that all partitions have weights proportional to their capacities. Let $(c_1, \ldots, c_k)$ be a vector of normalized relative partition capacities. Let $\mu := w_\Pi / k$ be the average partition weight. For a graph partitioning to be balanced according to the partition capacities it must hold that $\forall i \in \{1, \ldots, k\} \colon w_V(V_i) \leq (1 + \epsilon) \cdot \mu \cdot c_i$, where $\epsilon \in \mathbb{R}_{\geq 0}$ is a given imbalance parameter.

In the second case, we consider infrastructures with homogeneous nodes but **heterogeneous links** and sparse networks, i.e., networks that are not fully connected graphs. In this case, the mapping of partitions to nodes influences the communication costs. The goal of the mapping is to assign high communication volumes between partitions (i.e., cut edges with high weights) to wider links and short connection paths, ideally direct connections. Our strategy is to partition the workload graph without considering the infrastructure. We then use a greedy heuristic to repeatedly map the remaining partition with the highest total communication cost (wrt. the already mapped partitions) onto the node with the smallest distance.

The graph partitioning and mapping problem becomes even more complicated in presence of **heterogeneous nodes and a heterogeneous network**. Although the proposed strategies can be combined, the smaller degree of freedom makes it hard to find solutions. Given the restrictions in the fully heterogeneous case, we focus on systems that either have heterogeneous nodes or have a heterogeneous network.

Capacity Constraints. The infrastructure model distinguishes bounded and unbounded resources. To model both kinds of resources, we propose to add a *capacity constraint* to the already existing *balance constraint.* Partitioning algorithms can use the existing balance constraint with a carefully selected imbalance parameter to also enforce capacity constraints. A capacity constraint demands that all partition weights are below a given absolute capacity. Let $w_{\max}$ be a function that maps each partition to its maximum partition weight. For a graph partitioning to fulfill a capacity constraint it must hold that

$$\forall i \in \{1, \ldots, k\} \colon w_V(V_i) \leq w_{\max}(V_i).$$

Let the *total capacity* $w_{\max}(V)$ be the sum of all maximum partition weights. The *total requirement* equals the total partition weight w_Π of the graph.

To enforce the capacity constraint without restricting the solution space, a flexible imbalance parameter based on the available resources can be used. The imbalance parameter ϵ can be large when the total requirement is considerably smaller than the total capacity and approaches zero as the requirement approaches the capacity.

LEMMA 1. *For homogeneous nodes*[1]*, i.e,* $w_{max}(V_1) = \cdots = w_{max}(V_k)$*, a partitioning that fulfills a balance constraint with* $\epsilon = w_{max}(V)/w_\Pi - 1$ *also fulfills the capacity constraint. (The proof is straightforward and omitted due to space constraints.)*

Partial Allocations. As our last extension of the basic partitioning algorithm, we propose partial allocations. Application scenarios or service level agreements may require a subset of vertices to be pinned to certain nodes. The allocation problem then needs to consider this additional constraint. A special case occurs in presence of replication. Different replicas of the same object are commonly not allowed to share a node to ensure availability.

We enable both types of partial allocations, i.e., pinned vertices and *do not co-locate* constraints, by modifying the corresponding parts of the multilevel partitioning framework. Pinned vertices are assigned to their partitions before the actual partitioning starts. Furthermore, pinned vertices and vertices that lead to violations of location constraints when they are moved are not considered in the refinement methods. Note that partial allocations lead to fewer degrees of freedom and therefore possibly to suboptimal solutions or even cases with no valid solution at all.

5. MICRO-BENCHMARKS

For the implementation, we modified METIS (v5.1)[2] to support our penalized graph partitioning methods (we denote the resulting tool PENMETIS). METIS is a set of programs for graph partitioning and related tasks based on multilevel recursive bisection, multilevel k-way partitioning, and multi-constraint partitioning. Our modifications are based on the serial version of METIS but can also be incorporated in the parallel versions of METIS in the future.

In this section, we evaluate the overhead that penalized partition weights introduce in the partitioning process. Furthermore, we investigate how penalized graph partitioning scales with the size of the graph and the number of partitions. To analyze penalized graph partitioning, we use a exponential penalty function as example and graphs from the Walshaw Benchmark [28] . The corresponding Graph Partitioning Archive[3] contains 34 graphs from applications such as finite element computation, matrix computation, and VLSI design. The largest graph (`auto`) contains 448,695 vertices and 3,314,611 edges and can be considered a large workload graph.

In the first experiment, we investigate the **overhead of penalized partition weights**. Figure 4 shows the absolute partitioning times for all benchmark graphs using METIS and PENMETIS[4]. The figure shows that penalized graph

[1]A similar result can be obtained for heterogeneous nodes.
[2]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
[3]http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/
[4]We use a fairly moderate AMD Opteron (Istanbul) CPU running at 2.6 GHz for this experiment. As mentioned before, METIS and PENMETIS run single-threaded.

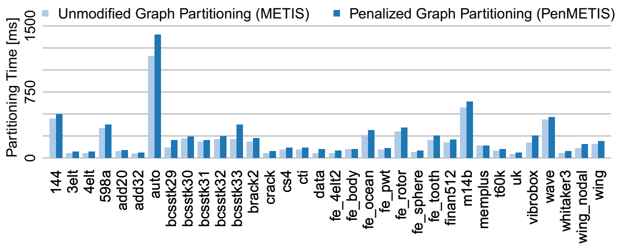


Figure 4: Partitioning Time Comparison (64 Partitions, 3% Imbalance)

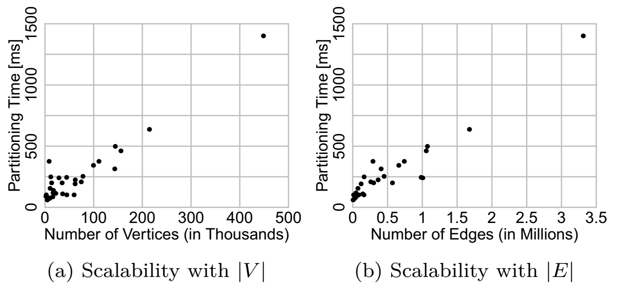


Figure 5: Execution Times of PenMETIS Charted by the Number of Vertices and Edges (64 Partitions, 3% Imbalance)

partitioning introduces only a small overhead. More specifically, PenMETIS takes on average 28% (42 ms) more time than METIS.

5.1 Scalability with Graph Size

To use graph partitioning with ever growing workload graphs, it is mandatory that the algorithms scale well with the size of the graph. Since penalized partition weights only induce little overhead, the scalability of PenMETIS is bound to the scalability of the underlying multilevel graph partitioning algorithms in METIS. Our second experiment investigates the **scalability with the size of the graph**. Figures 5a and 5b show the execution times of PenMETIS charted by the number of vertices and edges respectively. Our partitioning algorithm scales linearly with both parameters.

Our last scalability experiment investigates how **penalized graph partitioning scales with the number of partitions**. In Figure 6, we show partitioning times for METIS and PenMETIS for the largest benchmark graph and for various partition counts. Beyond 64 partitions, the partitioning time scales linearly with the number of partitions.

To summarize, our penalized graph partitioning shows the same performance and scalability behavior as the classic graph partitioning. While we presented the evaluation for an exponential penalty function, the performance is independent of the selected function and the same behavior can be seen for other penalty functions.

6. DBAAS EVALUATION

In this section, we present a full DBaaS system setup and evaluate our allocation strategy therein. We did experiments with different system sizes but only report the results for the largest system. In this case, we generated, deployed,

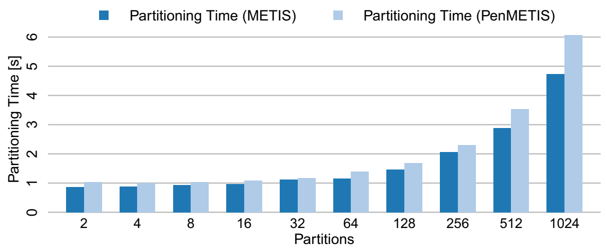


Figure 6: Scalability of Graph Partitioning with the Number of Partitions (Graph `auto`)

and queried 1,000 databases in our DBaaS system. The overall evaluation is done using our multi-tenancy database benchmark framework MulTe [18].

6.1 DBaaS System Setup

To evaluate our novel allocation strategy, we implemented a prototypical DBaaS system called *Multi-Tenancy-Middleware* (MTM)[5], which is similar to commercial DBaaS systems like Amazon RDS or Microsoft Azure. Databases in MTM can flexibly be provisioned on demand and the system takes transparently care of the physical representation of the databases. To support the analytical character of our DBaaS setup, we use the MTM system with a recent in-memory DBMS called ERIS as backend [19]. Given that the used in-memory DBMS supports only basic features, the raw performance is high and influences on the performance are not obscured along the system stack. Furthermore, with an in-memory DBMS, we can focus on the main memory as the primary resource to model.

6.2 Infrastructure Model

We conducted our experiments using Amazon Web Services. Multiple EC2 instances are used as back-end nodes. Additional EC2 instances are used to host the benchmark application, frontend tools, and the middleware. Our largest infrastructure setup consists of 32 instances as back-end nodes (`m4.xlarge`, 4 cores, 16 GiB memory) and 4 instances to run the benchmark (`c4.2xlarge`, 8 cores, 16 GiB memory).

Since processing is modeled as an unbounded resource, only relative capacities are needed in the infrastructure model. Given that all back-end nodes run on the same instance type, we assign a capacity of 1 (with respect to processing) to each node in the infrastructure graph. Additionally, each node has 16 GiB main memory which is added as the second (bounded) capacity to each node in the infrastructure graph. Furthermore, we assume a fully connected infrastructure graph with unit capacities on all links.

To investigate the non-linear resource usage of the infrastructure nodes, we conducted a number of synthetic experiments. As a result, we decided to model the non-linear resource usage with a penalty that is applied when more than 4 workload vertices share a node (note that each backend node contains 4 vCPUs). The penalty function is carefully handcrafted for the given setup and is best approximated with a function that is linear in the size of a small workload vertex for cardinalities between 5 and 20 before it starts to grow exponentially. Nevertheless, the determination of the penalty function is a challenging task and a topic of its own.

[5]https://goo.gl/zkOhpC

Final edited form was published in "UCC '16: 9th International Conference on Utility and Cloud Computing. Shanghai 2016", S. 200–209, ISBN 978-1-4503-4617-7
https://doi.org/10.1145/3006299.3006300

6.3 Workload Model

We use the Star Schema Benchmark (SSB) [24] as our analytical workload setting. Prior to our full DBaaS experiments, we evaluated the processing cost of each of the 13 SSB queries based on a cost model that uses the ERIS execution model [19]. Furthermore, databases are generated in 28 different sizes ranging from 10 MiB to 1 GiB. This results in 364 different basic database workloads with different workload intensities (28 database sizes times 13 queries in the benchmark; each database runs only a single query type). Together with a think time between queries, we are able to generate databases with any given intensity/cost.

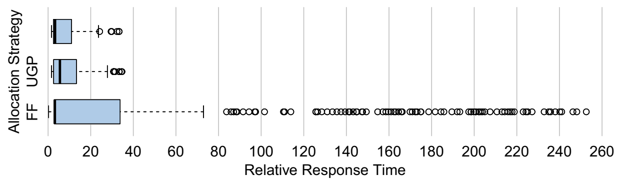
To run our DBaaS experiments, we generate a workload graph with random vertex weights and random edges. The graph consists of 1,000 vertices. The vertex weights represent processing costs and follow a Zipf distribution with exponent s = 1 ([27] observed similarly distributed tenants in an actual DBaaS system). Each vertex has a number of (randomly selected) neighbors. The number of neighbors also follows a Zipf distribution between 0 and 10. Neighbors (or edges in the workload graph) can result in distributed queries depending on the allocation. For each vertex, a database (i.e., a database size and a query type) is selected out of our pool that matches the vertex' weight best. A think time is added to match the database's processing cost exactly to the generated vertex weight.

6.4 Full System Evaluation

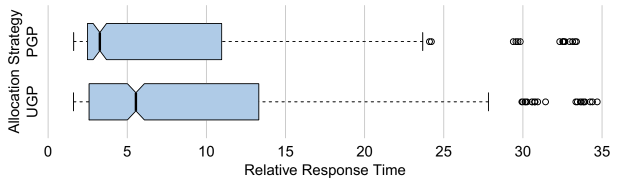
We compared three different allocation strategies in our experiments: First Fit (FF), Unmodified Graph Partitioning (UGP), and our Penalized Graph Partitioning (PGP). Distributing the databases (partitions) across all nodes using a round-robin strategy seems to be a simple approach to get a baseline. However, round-robin allocation does not consider memory capacities and this strategy might therefore produce invalid allocations where nodes are not able to keep all databases in memory. Instead, we use a greedy method (first fit) as a baseline allocation. This method sorts all databases by size (descending) and all nodes by utilization (ascending) and allocates each database to the least utilized node that is able to accommodate it. Using this strategy leads (in most cases) to valid allocations. Furthermore, using the least utilized node in each step tries to balance the load across all nodes.

The second allocation strategy (UGP) is based on the unmodified graph partitioning algorithm in METIS. Here, we partition the workload graph (without penalty) into 32 partitions to get a balanced allocation with minimal communication (32 nodes are available). The third allocation strategy (PGP) uses our penalized graph partitioning algorithm in PenMETIS. Our above mentioned penalty function is used to describe the infrastructure behavior in more detail and to produce better balanced allocation with minimal communication.

Performance Metrics. To evaluate the performance of the whole DBaaS system, we introduce the Relative Response Time (RRT). First, each absolute response time is transformed into a relative response time using a previously performed baseline run (i.e., the relative response time is relative to the best-case execution). Second, the 95% quantile of all relative response times for a given database is computed to get a single quality measure for each database. Using the quantile assumes that the user is interested in acceptable performance for the majority of executions.



(a) RRT Distribution, All Allocation Strategies



(b) RRT Distribution, Only Graph Partitioning Strategies

Figure 7: Relative Response Time Distribution (Note Different X-Axes)

Results. The experiments are executed with the following workflow. First, the setup is prepared based on the given allocation strategy. Once prepared, a baseline run is conducted where each database workload is executed individually to get best-case response times for each database. After the baseline run, the actual experiments run for 30 minutes. In these runs, the MulTe workload driver executes all workloads repeatedly and in parallel and collects execution statistics.

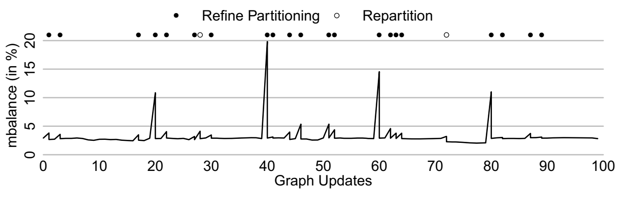
The results of our largest experiment (1,000 databases on 32 nodes) are summarized as box-whisker plots in Figure 7. Figure 7b shows the same results as Figure 7a on a differently scaled x-axis (UGP and PGP only). The plots in Figure 7a show that the first fit allocation strategy fails to balance the load, which leads to many outliers and a maximum Relative Response Time of 252. Comparing the allocation strategies that are based on graph partitioning, it can be seen in Figure 7b that Penalized Graph Partitioning leads to better overall system performance. PGP causes fewer outliers than UGP and has better maximum, average, and median RRTs. As a reference, computing the PGP allocation for the workload graph takes 6 ms using PenMETIS.

6.5 Incremental Update Experiment

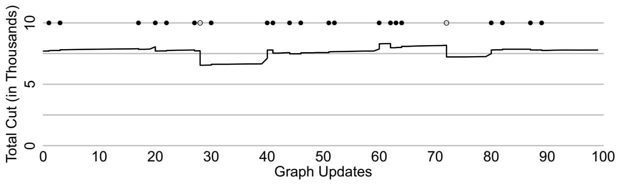
In this experiment, we evaluate the ability of our allocation strategies to react to changes in the workload. To simulate a changing workload, we define two workload graph modifications. A *minor change* is implemented by updating the vertex and edge weights of 1% of all vertices and all edges. A *major change* is implemented by updating the vertex and edge weights of 10% of all vertices and all edges. The complete experiment consists of 100 workload changes. After every 20 minor changes, one major change is simulated. The results are shown in Figure 8.

After each workload change, the current partitioning is evaluated against the new workload graph. The update mechanism is triggered when the balance constraint is violated. The update strategy first tries to regain a balanced partitioning by using local refinement strategies. A com-

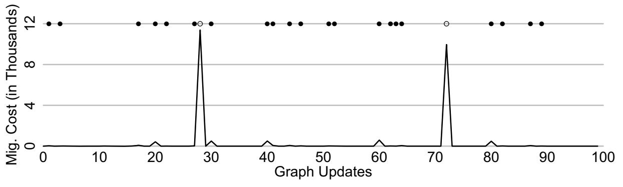
Final edited form was published in "UCC '16: 9th International Conference on Utility and Cloud Computing. Shanghai 2016", S. 200–209, ISBN 978-1-4503-4617-7
https://doi.org/10.1145/3006299.3006300



(a) Imbalance of the Graph in Percent (Before and After Refinements/Repartitionings)



(b) Total Cut of the Graph in Thousands (Before and After Refinements/Repartitionings)



(c) Migration Cost for the Refinements/Repartitionings

Figure 8: Incremental Update Experiment (32 Partitions, 3% Imbalance)

plete repartitioning is only triggered when the local refinement fails. In addition, the update strategy repartitions the workload graph in the background after every ten changes. However, the new partitioning is only implemented when it leads to a total cut that is better by more than 10% of the old cut. The evolutions of the graph imbalance and the total cut are summarized in Figures 8a and 8b. The results show that minor changes eventually and major changes always lead to violations of the balance constraint. However, in many cases (21 out of 23 in the experiment) the local refinement algorithm is able to regain a balanced partitioning. A complete repartitioning is triggered only in two cases, which in both cases leads to considerably better total cuts.

We report the sum of all vertex weights of vertices that are moved between partitions as the total migration cost for an update. The migration costs are shown in Figure 8c. The figure shows that partitioning the workload graph from scratch causes considerably higher migration costs than refining an existing partitioning.

7. RELATED WORK

Graph partitioning has been a topic of interest in the high performance computing community at least since the late 1990s. Early works on the multilevel graph partitioning paradigm [15] led to many papers about variations and extensions of the balanced min-cut partitioning problem, e.g., about multi-constraint partitioning [16] or incremental update strategies [13]. A rather recent survey provides an excellent overview of the results in the field [5]. To the best of our knowledge, we are the first to consider non-linear graph partitioning.

Several authors have proposed variations of the allocation problem. Each formulation uses different objectives and constraints as well as different assumptions on workload and infrastructure. As part of the Relational Cloud Project[6], Curino et al. investigate workload-aware database monitoring and consolidation [7]. The presented Kairos system consolidates databases based on the predicted combined resource utilization. In the Kairos system, CPU, memory, and I/O load are modeled and non-linear behavior is captured in a *combined load predictor* that is used for combining I/O load. A linear combination of the three resources, possibly weighted to indicate the relative importance, is used in the objective function of the consolidation problem. Kairos does not consider distributed databases and hence completely omits communication costs. A second major difference is that Kairos tries to minimize the number of nodes with the constraint of not overcommitting any node.

Schism [9] by Curino et al. is also part of the Relational Cloud Project. Schism and our work share the idea of using graph partitioning methods for the allocation problem. Both works model the workload as a weighted graph and seek balanced partitions that minimize the communication. The infrastructure model in Schism is simpler compared to ours, non-linear hardware is not considered. The authors discuss balancing data size or data accesses. However, the presented solution has no notion of bounded or unbounded resources and all balanced partitionings are considered valid. Schism assumes a static workload and does not provide means to incrementally update a partitioning.

Quamar et al. picked up the idea of Schism and present a project called SWORD [26]. The authors propose a number of techniques to achieve higher scalability and to increase tolerance in presence failures and workload changes. In a follow-up [20], the authors introduce the notion of the *query span*, i.e., the average number of machines involved in the execution of a query. SWORD does not consider non-linear hardware.

Schaffner et al. introduced the Robust Tenant Placement and Migration Problem (RTP) [27]. Unlike our allocation problem, the RTP tries to minimize the number of servers required by consolidating in-memory databases. The authors make the case for incremental tenant placement. Migration costs are modeled and quantified in the RTP and performance is guaranteed even while a database is being migrated. Distributed databases are intentionally not considered in the RTP and communication costs are omitted with the exception of migration costs.

Lang et al. [21] investigate Service Level Objectives (SLO) in the context of multi-tenant DBaaS systems. Their goal is to minimize the operational cost while being SLO compliant. The authors investigate a small number of classes of SLOs and constant workload. Thereby, the authors experimentally determine a non-linear performance behavior in the heterogeneous case (comparable to our idea of a penalty function) and propose a brute-force solver for the non-linear integer programming problem. However, it is not obvious how the approach translates to an arbitrary number of different workloads. Furthermore, communication between tenants is not considered in [21].

Liu et al. [23] propose approximation algorithms for the tenant placement problem with the goal to maximize profit

[6]http://relationalcloud.com/

under given SLA penalties. The authors use a server load that is a combination of a base load and a factor that accounts for the increase of execution time when a set of tenants is packed on the server (comparable to our penalty). In their experiments, the authors determine this factor for a single workload type. Like previous approaches, Liu et al. do not consider communication costs or dynamic placement of tenants.

8. CONCLUSION

In this paper, we used weighted graphs to model DBaaS systems and formulated the allocation problem based on a generalized workload and infrastructure. Furthermore, we presented our penalized graph partitioning as an allocation strategy for DBaaS systems. Thereby, we relaxed limitations of the basic method and enabled incremental updates. An extensive experimental evaluation showed the applicability and scalability of our penalized graph partitioning as well as the performance benefit in an actual DBaaS system. We believe that our penalized graph partitioning is a versatile method that can be applied to many other domains.

We are aware of the many challenges to overcome in the process of finding good performance models, i.e., penalty functions. Here, we presented a solution that solves the allocation problem under the assumption of the penalized performance model. A systematic evaluation of experimental and incremental methods to find penalty functions is subject to future work.

9. REFERENCES

[1] M. Ahmad and I. T. Bowman. Predicting System Performance for Multi-Tenant Database Workloads. In *DBTest*, 2011.

[2] K. Andreev and H. Räcke. Balanced graph partitioning. In *SPAA*, pages 120–124, 2004.

[3] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-Aware Scheduling on Multicore Systems. *ACM Transactions on Computer Systems*, 28(4), 2010.

[4] S. Blagodurov, S. Zhuravlev, A. Fedorova, and M. Dashti. A Case for NUMA-Aware Contention Management on Multicore Systems. In *PACT*, 2010.

[5] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. *preprint: Computing Research Repository*, 2013.

[6] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozda, R. Heaphy, and L. A. Riesen. Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations. In *IPDPS*, 2007.

[7] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan. Workload-Aware Database Monitoring and Consolidation. In *SIGMOD*, 2011.

[8] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database-as-a-Service for the Cloud. In *CIDR*, 2011.

[9] C. Curino, E. P. C. Jones, Y. Zhang, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. In *VLDB*, 2010.

[10] K. D. Devine, E. G. Boman, R. T. Heaphy, and B. A. Hendrickson. New Challenges in Dynamic Load Balancing. *Applied Numerical Mathematics*, 52, 2005.

[11] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *DAC*, 1982.

[12] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of "big data" on cloud computing: Review and open research issues. *Inf. Syst.*, 47:98–115, 2015.

[13] B. Hendrickson, R. Leland, and R. Van Driessche. Enhancing Data Locality by Using Terminal Propagation. In *HICSS*, 1996.

[14] L. Hyafil and R. L. Rivest. Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems. Technical report, IRIA, 1973.

[15] G. Karypis and V. Kumar. Analysis of Multilevel Graph Partitioning. In *SC*, 1995.

[16] G. Karypis and V. Kumar. Multilevel Algorithms for Multi-Constraint Graph Partitioning. Technical report, University of Minnesota, Department of Computer Science, 1998.

[17] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49(2), 1970.

[18] T. Kiefer, B. Schlegel, and W. Lehner. MulTe: A Multi-Tenancy Database Benchmark Framework. In *TPCTC*, 2012.

[19] T. Kissinger, T. Kiefer, B. Schlegel, D. Habich, D. Molka, and W. Lehner. ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workloads. In *ADMS*, 2014.

[20] K. A. Kumar, A. Quamar, A. Deshpande, and S. Khuller. SWORD: Workload-Aware Data Placement and Replica Selection for Cloud Data Management Systems. *The VLDB Journal - The International Journal on Very Large Data Bases*, 23(6), jun 2014.

[21] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards Multi-Tenant Performance SLOs. In *ICDE*, 2012.

[22] C. Li, C. Ding, and K. Shen. Quantifying the Cost of Context Switch. In *ExpCS*, 2007.

[23] Z. Liu, H. Hacigümüs, H. J. Moon, Y. Chi, and W.-P. Hsiung. PMAX : Tenant Placement in Multitenant Databases for Profit Maximization. In *EDBT*, 2013.

[24] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*, 2009.

[25] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-Oriented Transaction Execution. In *VLDB*, 2010.

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

[27] J. Schaffner, T. Januschowski, M. Kercher, T. Kraska, H. Plattner, M. J. Franklin, and D. Jacobs. RTP: Robust Tenant Placement for Elastic In-Memory Database Clusters. In *SIGMOD*, 2013.

[28] A. J. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning. *Journal of Global Optimization*, 29(2), 2004.