TECHNISCHE UNIVERSITÄT DRESDEN

Allocation Strategies for Data-Oriented Architectures

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Inf. Dipl.-Math. Tim Kiefer
geboren am 25. September 1983 in Saalfeld

Gutachter:

Prof. Dr.-Ing. Wolfgang Lehner
Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden

Prof. Johann-Christoph Freytag, Ph.D.
Humbold-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik
Lehrstuhl für Datenbanken und Informationssysteme
Unter den Linden 6, 10099 Berlin

Tag der Verteidigung: 9. Oktober 2015

Dresden, Oktober 2015

Abstract

Data orientation is a common design principle in distributed data management systems. In contrast to process-oriented or transaction-oriented system designs, data-oriented architectures are based on data locality and function shipping. The tight coupling of data and processing thereon is implemented in different systems in a variety of application scenarios such as data analysis, database-as-a-service, and data management on multiprocessor systems. Data-oriented systems, i.e., systems that implement a data-oriented architecture, bundle data and operations together in tasks which are processed locally on the nodes of the distributed system. Allocation strategies, i.e., methods that decide the mapping from tasks to nodes, are core components in data-oriented systems. Good allocation strategies can lead to balanced systems while bad allocation strategies cause skew in the load and therefore suboptimal application performance and infrastructure utilization. Optimal allocation strategies are hard to find given the complexity of the systems, the complicated interactions of tasks, and the huge solution space. To ensure the scalability of data-oriented systems and to keep them manageable with hundreds of thousands of tasks, thousands of nodes, and dynamic workloads, fast and reliable allocation strategies are mandatory.

In this thesis, we develop novel allocation strategies for data-oriented systems based on graph partitioning algorithms. Therefore, we show that systems from different application scenarios with different abstraction levels can be generalized to generic infrastructure and workload descriptions. We use weighted graph representations to model infrastructures with bounded and unbounded, i.e., overcommited, resources and possibly non-linear performance characteristics. Based on our generalized infrastructure and workload model, we formalize the allocation problem, which seeks valid and balanced allocations that minimize communication. Our allocation strategies partition the workload graph using solution heuristics that work with single and multiple vertex weights. Novel extensions to these solution heuristics can be used to balance penalized and secondary graph partition weights. These extensions enable the allocation strategies to handle infrastructures with non-linear performance behavior. On top of the basic algorithms, we propose methods to incorporate heterogeneous infrastructures and to react to changing workloads and infrastructures by incrementally updating the partitioning.

We evaluate all components of our allocation strategy algorithms and show their applicability and scalability with synthetic workload graphs. In end-to-end–performance experiments in two actual data-oriented systems, a database-as-a-service system and a database management system for multiprocessor systems, we prove that our allocation strategies outperform alternative state-of-the-art methods.

Acknowledgments

My gratitude first and foremost goes to my advisor Wolfgang Lehner. His enthusiastic lectures made me choose my specialty, his advertisement led to my first internship at IBM, and his support made my diploma thesis and the following scholarship possible. Wolfgang kept me on track in the last few years and repeatedly reminded me how awesome my topic is and how thrilling the research around it can be. I remember countless discussions, all following the same script of me being skeptical towards my topic and my results and him pointing out the endless research opportunities. Thank you for believing in me!

I am grateful to Johann-Christoph Freytag for co-advising my thesis and for his many valuable hints. I would like to thank the Deutsche Telekom Stiftung for making this thesis possible, Klaus Kinkel for being all-in on MINT education, Christiane Frense-Heck for organizing everything and solving every problem, and all my fellow scholarship holders for great fun at all the events and meetings.

Special thanks go to Benjamin Schlegel. He sparked my interest in the modern-hardware aspect of my work and offered an alternative when my topic seemed to be a dead end. I would also like to thank Thomas Kissinger for our fruitful collaboration on the NUMA topic. New York made it all possible...

I would like to thank Dirk Habich for reading major parts of the thesis and for the many valuable hints that followed in long discussions. My gratitude for proofreading goes furthermore to Claudio Hartmann und Stefanie Gahrig.

I am thankful to all my past and current colleagues for creating an inspiring and fun working atmosphere. Martin Hahmann and Katrin Braunschweig were awesome roommates and I enjoyed our many research-related and off-topic discussions.

All this work would not have been possible without my family and friends. My parents made me who I am today and I will always be grateful for that. Special thanks go to my beloved partner Ilka and my daughter Alexandra. Ilka always had my back, believed in me, encouraged me, and never doubted my eventual success. Alexandra did her part by sleeping through virtually every night. Thank you! You are truly awesome!

Tim Kiefer
Dresden, July 13, 2015

Contents

1. Introduction

The term *data-oriented architecture* is not fixed and does not refer to a single system layout. In the context of data management systems, data orientation is commonly used to describe distributed data processing systems that make the data a first class citizen and drive the processing based on data locality. Unlike process-oriented or transaction-oriented architectures, where a single process is assigned to acquire the necessary data, perform the operations, and return the results, in data-orientated architectures the data drives the processing. Figure 1.1 shows the two different execution strategies, data-orientated and process-orientated, side by side. In data-oriented systems, i.e., systems that implement a data-oriented architecture, transactions[1] are split into self contained units of work. Data and the operations performed thereon are bundled in *tasks* and tasks are processed locally on the *nodes* of the distributed system. The term *allocation* in data-oriented systems refers to the mapping of tasks to nodes. Data locality is a key design principle in data-oriented systems, i.e., tasks are executed where the data resides instead of moving the data to the processing. In other words, a data-oriented architecture uses function shipping while process-oriented architectures are based on data shipping. Data can still be moved in data-oriented systems, but only explicitly when tasks communicate to coordinate or perform complex workloads.

Data orientation is a key design principle in various data management systems with very different abstraction levels. The following three exemplary application scenarios use data-oriented systems:

Data Analysis Several data analysis platforms were developed in recent years. Examples are MapReduce at Google, the open-source alternative Hadoop, or Microsoft's Cosmos. Advanced data processing capabilities, e.g., SQL processing, are either integral part of the system (e.g., Scope in Cosmos, Zhou et al., 2012) or have been added on top (e.g., Impala on Hadoop, Impala, 2015). Data in these distributed analysis platforms is always processed locally and only intermediate results are transferred between tasks.

Database-as-a-Service Providers like Amazon (Amazon, 2015) or Microsoft (Microsoft, 2015) offer relational databases as a service in their respective cloud infrastructures. A relational database (task) comprises the actual data and the operations, i.e., SQL statements. Databases are stored on a single server (node) or distributed across multiple servers.

Database Management on Multiprocessor Systems Database management systems (DBMSs) on modern multiprocessor systems face the challenge of efficiently utilizing the available resources. Cores and memory controllers are

[1] The term *transaction* here refers to any query, statement, or data-processing code

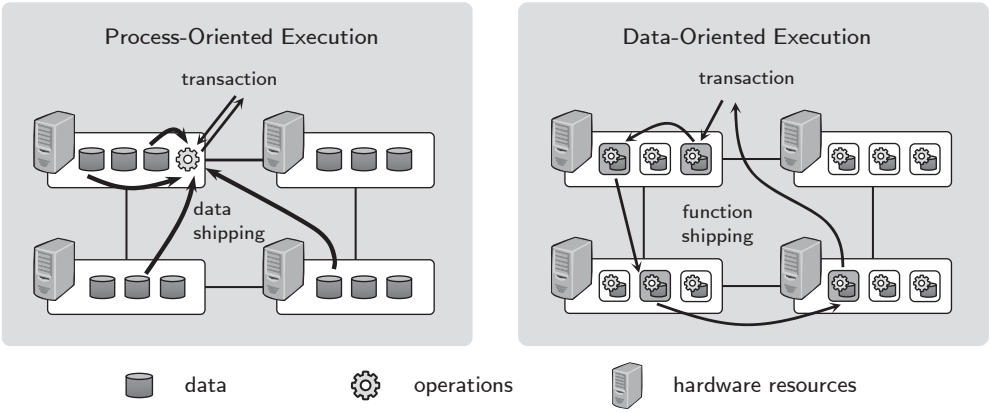


Figure 1.1.: Execution Schemes in Distributed Data Processing Systems

distributed among the processors. The best performance can be achieved by treating the multiprocessor system as a distributed system. Optimized storage engines like the one in the in-memory DBMS ERIS execute portions of a query locally on a single core (Kissinger et al., 2014). Parts of the query execution plan and the involved data form a task that is assigned to a multiprocessor (node). Only intermediate data is transferred between tasks if necessary.

In this thesis, we investigate allocation strategies for systems that implement data-oriented architectures in general. We demonstrate the applicability of our solutions in two exemplary systems from the second and the third category, respectively. Although the two scenarios are very different, they base on common principles and we show that a unified allocation strategy can be used for both.

Given that the location of data determines where processing is performed, the allocation is crucial in any data-oriented system. The allocation can lead to a balanced or a severely skewed system and has therefore several implications on global system characteristics, e.g., the utilization of the nodes and the performance of the tasks. Consolidation, i.e., co-location and concurrent execution of (possibly unrelated) tasks on single nodes is a central means to optimize utilization, hence cost-effectiveness, and performance in data-oriented systems (Curino et al., 2011). Consequently, the decision on where data resides in data-oriented systems, i.e., the allocation strategy, is commonly taken from the user and optimized by the system. This optimization is workload-driven, i.e., based on the actual tasks. Figure 1.2 shows an abstract data-oriented architecture. Knowledge about both the tasks of the workload and their characteristics as well as the infrastructure is used by an allocation strategy that decides the initial data layout as well as periodic data migrations, which are required to balance the load across the nodes. The location of the tasks is transparent to the users of actual data-oriented systems.

Finding an optimal allocation is an inherently difficult problem. The allocation actually comprises two problems, (1) determining a partitioning of the data, i.e., a granularity of the tasks, and (2) finding a mapping of tasks to nodes. Even on a small scale, solving the allocation problem is hard, given the complex interactions

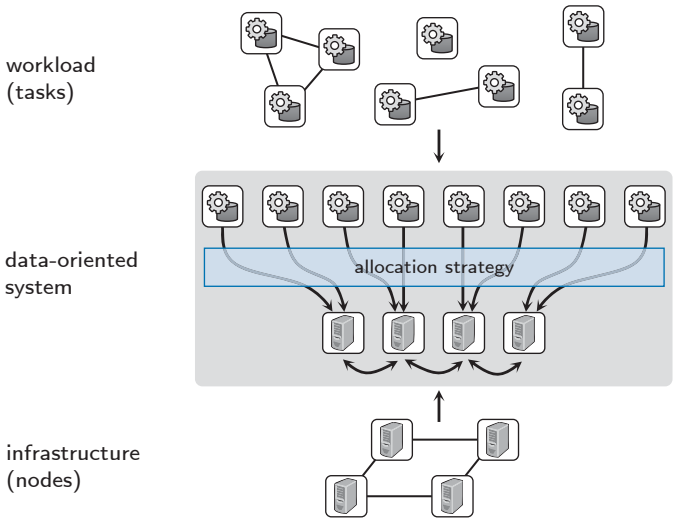


Figure 1.2.: Data-Oriented Architecture

that tasks may have (Ahmad and Bowman, 2011). Contention on hardware or operating system resources may lead to performance characteristics that are hard to predict. The presence of heterogeneous hardware, dynamic workloads, and dynamic infrastructures add to the complexity of the problem. To ensure the scalability of data-oriented systems and to keep them manageable with hundreds of thousands of tasks and thousands of nodes, fast and reliable algorithms for the allocation problem are mandatory. However, distributed data processing systems and complex workloads are hard to model and system behavior is hard to predict (Ahmad and Bowman, 2011; Curino et al., 2011). Even with the models, the allocation problem has a huge solution space and the problem itself is NP-hard (e.g., Curino et al., 2011; Schaffner et al., 2013).

An ideal allocation strategy can be used for optimizing different objectives like minimizing the required number of nodes or maximizing task performance. Furthermore, it is scalable to thousands of nodes and hundreds of thousands of tasks, flexibly reacts to dynamically changing workloads and infrastructures, and deals with complex, heterogeneous components.

Summary of Contributions

The contribution of this thesis is a set of allocation strategies for data-oriented systems. We thereby assume an existing partitioning and focus on the mapping part of the allocation problem. Our allocation strategies model workload and infrastructure information in weighted graphs and solve the allocation problem based on graph partitioning and mapping algorithms. In detail, our contributions are:

- We analyze data-oriented systems in different application scenarios, specifically database-as-a-service systems and DBMSs for multiprocessor systems. We identify core characteristics of the systems and abstract them to a generic data-oriented system. We formulate the allocation problem based on the abstract data-oriented system.

- We abstract from the infrastructures in the example systems to a generic graph-based representation of available resources. Thereby, we account for non-linear performance caused, e.g., by contention. Furthermore, we introduce the notion of bounded and unbounded resources to be able to model commonly oversubscribed resources and resources that render an allocation invalid if overused.
- We abstract from the workloads in the example systems to a generic graph-based representation of the resource requirements. Each task can consume multiple resources that will be treated individually by the allocation strategy.
- We present allocation strategies based on balanced k-way min-cut graph partitioning. We extend known heuristics for graph partitioning and multi-constraint graph partitioning by methods to account for non-linear resource behavior. Thereby, we introduce the notion of *penalized weights* and *secondary weights*. We show how to balance these penalized and secondary weights during graph partitioning. To the best of our knowledge, our method is the first to partition a graph with vertex weights that do not combine linearly to partition weights.
- We show that our basic allocation strategies can be extended to fulfill different requirements. We present ways to modify the allocation strategy to account for heterogeneous hardware. Furthermore, we show how an allocation can incrementally be updated after changes to handle dynamic workloads and infrastructures.
- We experimentally evaluate our method using synthetic scenarios. In addition, we implement our allocation strategies in two actual data-oriented systems and evaluate their benefit in end-to-end–performance experiments.

Outline

The structure of this thesis is illustrated in Figure 1.3. Chapter 2 presents foundations of data-oriented systems with a focus on database-as-a-service systems and DBMSs for multiprocessor systems. Different implementation strategies and properties of the corresponding systems are described together with existing solutions in this field. Chapter 3 abstracts from actual systems and details all necessary components of the allocation problem before the problem itself is formulated. First, the infrastructure and the workload model used in the allocation problem are abstracted from actual systems. Second, the allocation problem is defined and variations are discussed. Third, requirements for allocation strategies are derived from the proposed models and the problem formulation.

Chapters 4 and 5 present allocation strategies, i.e., solutions for the allocation problem, based on the abstract data-oriented system. Chapter 4 details our allocation strategies, which are based on balanced k-way min-cut graph partitioning. Starting from existing graph partitioning algorithms, we propose new methods for graph partitioning with penalized and secondary weights. In Chapter 5, we extend and modify the graph partitioning algorithms to relax previously made assumptions. Details are provided that make the allocations strategies applicable to heterogeneous

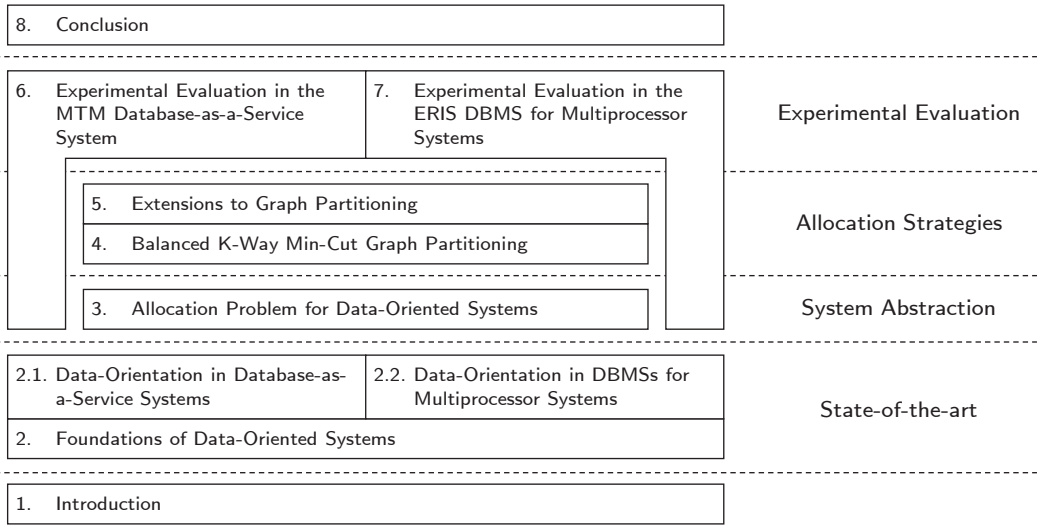


Figure 1.3.: Structure of this Thesis

infrastructures and systems with dynamic workloads and infrastructures. All aspects of the allocations strategies are experimentally evaluated in the corresponding chapters.

The allocation strategies in their entirety are experimentally evaluated in two actual data-oriented systems in Chapters 6 and 7. Chapter 6 presents the database-as-a-service system MTM and evaluates our allocation strategies in this system. In Chapter 7, we analyze properties of multiprocessor systems and introduce ERIS, a DBMS that is optimized for multiprocessor systems. Furthermore, we show details of the implementation and evaluation of our allocation strategies in ERIS.

We finally conclude this thesis in Chapter 8 with a summary and a discussion of future work.

2. Foundations of Data-Oriented Systems

In this chapter, we present foundations of data-oriented systems to provide necessary basics for the subsequent chapters. Related work that is relevant to the topic will be presented throughout the three parts of this chapter. We first present two application scenarios that implement data-oriented architectures in Sections 2.1 and 2.2. The two scenarios result in two classes of data-oriented systems, namely *database-as-a-service systems* and *DBMSs for multiprocessor systems*. A variety of actual systems exist in both classes that differ in the implementation or in the degree of data orientation, i.e., the granularity of tasks and the degree of data locality. By giving an overview of these systems, we show that although being different in many aspects, they follow common principles that allow for a unified allocation strategy. Two example systems, one from each class, are described and evaluated in detail in the experiments Chapters 6 and 7.

Second in this chapter, in Section 2.3, we regard the relation of data-oriented systems to classic distributed relational DBMSs and revisit related work in this field. Distributed DBMSs have been studied at least since the late 1980s and some of the early problems sound surprisingly familiar to our allocation problem. For instance, Apers (1988) formulated almost 30 years ago the following problem: "Given the queries and updates, the frequencies of their usage, and the sites where the results have to be sent, determine (1) the fragments to be allocated, and (2) allocate these fragments, possibly redundant, and the operations on them to the sites of the computer network such that a certain cost function is minimized." While many principles of distributed DBMSs apply to the data-oriented systems in this thesis, the latter add several challenges. Existing commercial distributed DBMSs provide the means for partitioning and distributed processing of data but keep the decisions on partitioning and placement to the domain expert. We aim for system sizes that make an automated and possibly fine-grained allocation strategy mandatory. Furthermore, the main focus of distributed DBMSs has traditionally been to scale out huge database applications that would either not fit on a single node or benefit from the parallel data processing. The goal of many data-oriented systems is to also provide and optimize scale-in scenarios to economically run a large number of tasks on a small number of nodes. At last, allocation strategies in data-oriented systems must be able to deal with heterogeneous infrastructures and dynamic workloads, conditions not always found in distributed DBMS scenarios. Nevertheless, related work especially on physical database design advisors and work on automated partitioning and partition placement provide helpful insights and will be presented in the second part of this chapter.

Two more topics that are important foundations of this thesis are not discussed in this chapter, but later in the thesis. Related work on the allocation problem is best understood after preliminaries like the infrastructure and workload model have been

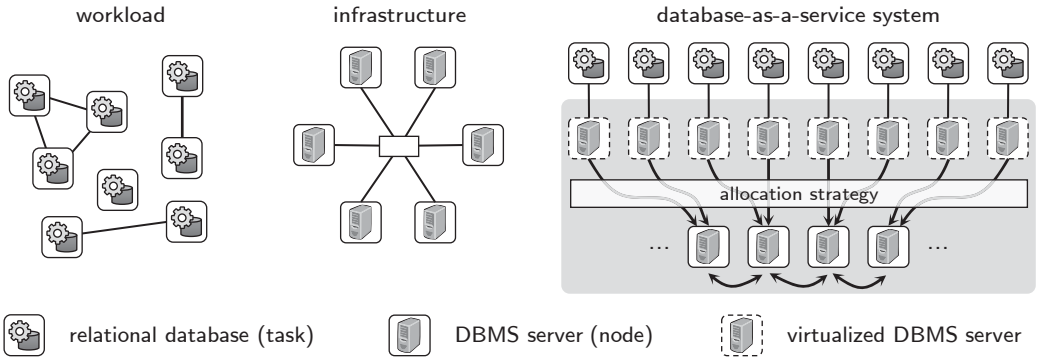


Figure 2.1.: Database-as-a-Service Platform Depicted as Data-Oriented System

detailed and will hence be presented in Section 3.5 in the next chapter. Foundations of graph partitioning will be given in the context of our new partitioning methods in Chapter 4.

2.1. Data Orientation in Database-as-a-Service Systems

Cloud computing has become a valid usage model for different kinds of applications and a successful business model for several service providers. Using a resource, e.g., the infrastructure (Infrastructure-as-a-Service, IaaS), an application platform (PaaS), or the software itself (SaaS), as a service has several advantages. The resource can be used flexibly and it is scalable to accommodate different or shifting application needs. Resources can be quickly provisioned in the cloud to offer applications a short time to market. The shift from capital expenditure to operational expenditure and the pay-as-you-go billing model foster start-ups and experimental applications. For service providers, the economy of scale, enabled by resource sharing, renders cloud computing interesting.

Offering (relational) databases as a service transfers the advantages of cloud computing to the data storage layer of an application and allows applications that rely on a storage layer to run in the cloud. All major cloud service providers offer relational databases as part of their product portfolio (Amazon, 2015; Microsoft, 2015; Oracle, 2015).

2.1.1. Allocation Problem in Database-as-a-Service Systems

Database-as-a-service systems implement data-oriented architectures, the analogies are shown in Figure 2.1. Relational databases comprise the actual data in form of tables and the operations thereon in form of SQL statements. Hence, relational databases form the tasks in the data-oriented system where database objects represent the data. Relational databases can be partitioned which leads to tasks that communicate to execute a workload. The nodes of the system are connected database management system servers, which together build the infrastructure.

Conceptual, the database-as-a-service system offers a virtual DBMS server wherein the service customers create and use their databases. These virtual DBMS

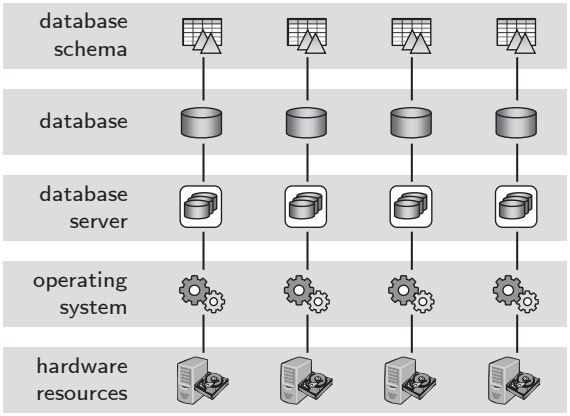


Figure 2.2.: Database System Stack

servers are internally mapped to actual server machines. Which layers of the system stack belong to the abstract task and which to the abstract node depends on the actual implementation of the system and will be detailed in the next section.

The allocation problem is of high interest in database-as-a-service systems (e.g., Curino et al., 2011; Schaffner et al., 2013). The allocation strategy can for instance be used to minimize the number of nodes and thereby reduce resource consumption and operational costs. Furthermore, the allocation strategy can help to improve performance, to guarantee service levels, and to ensure availability.

2.1.2. Implementation Classes of Database-as-a-Service Systems

Database-as-a-service systems are implemented by virtualizing the database server and consolidating multiple virtual servers on a single physical server. However, the multi-layered system stack of a database management system, ranging from the hardware layer up to the database schema (shown in Figure 2.2) allows for virtualization of different levels with different characteristics of the resulting system. Understanding the individual characteristics is mandatory to be able to extract the necessary workload and infrastructure information to solve the allocation problem (details follow in Chapter 3). Especially, understanding how resources are monitored, concurrently used by tasks, and isolated between tasks is important for the infrastructure model. Assuming that only one layer is virtualized in any solution, the stack leads to the five classes that are shown in Figure 2.3 (Hui et al., 2009; Jacobs and Aulbach, 2007; Kiefer and Lehner, 2011; Wang et al., 2008):

1. Private OS (Figure 2.3 on the left),
2. Private Process,
3. Private Database,
4. Private Schema, and
5. Shared Schema (Figure 2.3 on the right).

The data-oriented systems differ among the implementation classes. Especially, the granularity of tasks and nodes depends on the layer that is virtualized and

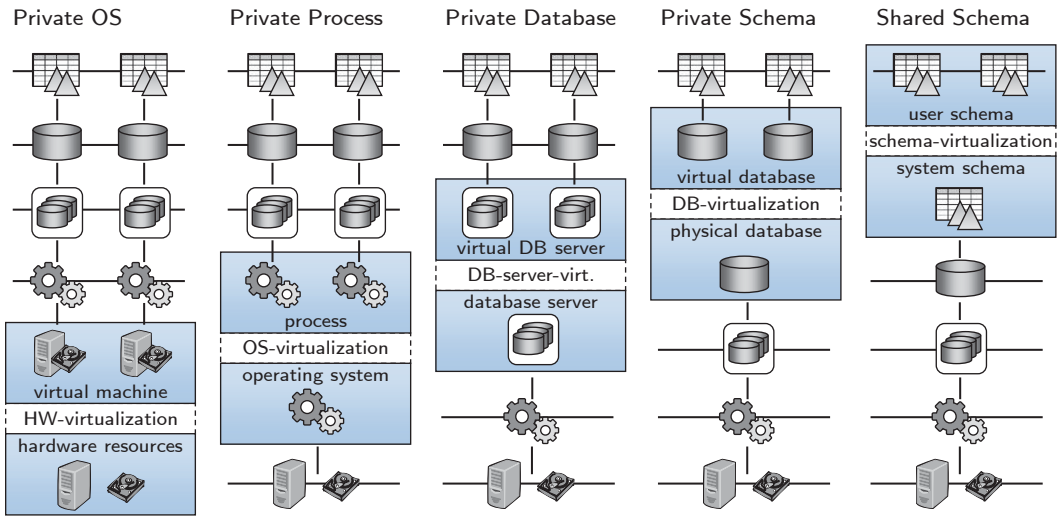


Figure 2.3.: Classification of Database-as-a-Service Systems

consequently on the portion of the system stack that is shared and the portion that is private. Figure 2.4 exemplarily illustrates what comprises a task and what comprises a node in the Private OS and the Private Schema implementation classes, respectively. In general, layers of the system stack above the virtualized layer belong to the task (and are described in the workload model) whereas layers of the systems stack below the virtualized layer belong to the node (and are hence described in the infrastructure model).

Private OS

With Private OS, hardware is virtualized with the help of virtual machines. Each database application, i.e., each task, comprises the stack from the operating system upward. Hence, for each task the platform has to deploy and maintain an operating system and a database management server with the respective database. This class offers the highest degree of isolation with respect to security, performance, and availability. At the same time, the resource requirements per database are the highest which leads to a rather small number of tasks that can share a single node.

Resource usage of each task can be monitored in the database system, in the guest operating system, or by means of the virtual machine monitor.

Private Process

In the Private Process class, the operating system is virtualized. Each task is represented by a database server process and several such processes share the operating system. Since the operating system is only needed once, this class can support a higher number of tasks per node. However, a large number of concurrent processes may harm the system performance and the individual processes must request resources responsibly, e.g., only the amount of memory that they actually need.

With private processes, operating system facilities can be used to monitor and isolate certain system resources. The operating system scheduler assigns CPU time

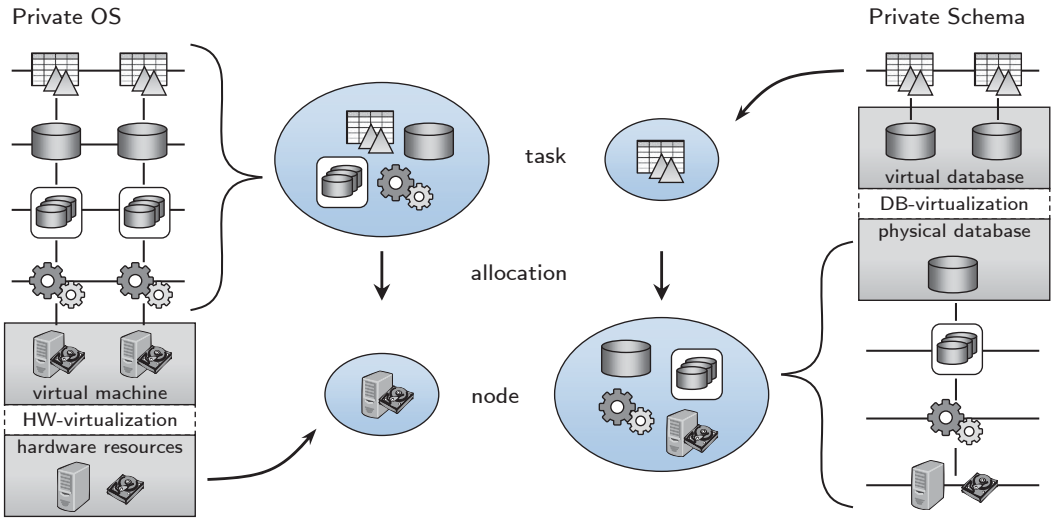


Figure 2.4.: Task and Node Granularity in Database-as-a-Service Systems

to the various competing processes and priority levels and sometimes quotas can be used to increase or decrease the share of any process. The main memory is per default not limited by the operating system. Each process can use the same virtual address space. The operating system only takes care of mapping the virtual memory to physical memory and of paging in case of memory shortage.

Lightweight application containers like Docker (Docker, 2015) can be used to further control execution of and increase isolation between the various tasks.

Private Database

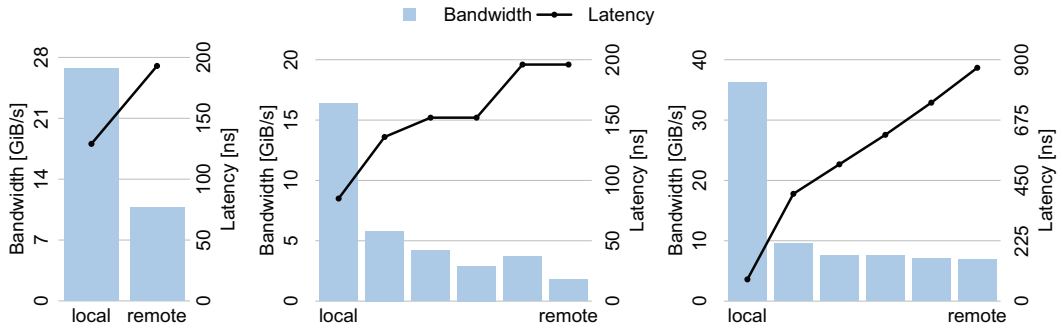
The third system class shown in Figure 2.3 implements private databases. Here, a single server process hosts a number of private databases, i.e., tasks.

The database management system needs to provision resources and balance loads between different tasks. The different tasks share a database process and hence are usually indistinguishable by the operating system. The only way to isolate the tasks' performances is by means of managing and isolating DBMS resources. In our experience, buffers (e.g., for pages or sorts) can usually be split and assigned to different databases. Other resources, such as the logging facility, are usually shared and can hence lead to contention.

The detailed isolation options in currently available systems depend on the DBMS implementation and are usually very limited. In the research community, Das et al. (2014) and Narasayya et al. (2013) investigate the problem of performance isolation in shared-process database-as-a-service systems. They present SQLVM, an abstraction for performance isolation in the DBMS. Furthermore, they implement and test a prototype of SQLVM in Microsoft Azure.

Private Schema

With Private Schema, the database itself is virtualized. Each task can access a virtual database, i.e., a schema, that is mapped to an actual physical database.



(a) 4-Node Machine (b) 8-Node Machine (c) 64-Node Machine

Figure 2.5.: Memory Access Benchmark Results

Several schemas can share a single database, which leads to a very low overhead per task (schemas can be added to a database almost indefinitely).

The private schema class is similar to the private database class with respect to the performance isolation characteristics. However, some resources like page buffers may be even harder or impossible to isolate in a shared database.

Shared Schema

In the Shared Schema class, database applications share all layers of the database system stack. The database schema is virtualized such that each task sees a private schema, which is internally mapped to a single system schema (for schema mapping techniques refer to, e.g., Aulbach et al., 2009, 2011).

This class provides the least isolation, because even access structures like indexes are shared. However, the resource overhead per task is the smallest among all classes. This class lends itself to applications where different users share a common (or similar) schema. The loss of any separation of the databases leads to high maintenance costs for operations such as backup or migration.

2.1.3. Summary and Discussion

Concluding this section, there is a variety of possible implementations for database-as-a-service systems. Different characteristics and tradeoffs lead to valid use-cases for all classes. Depending on the implementation, the workload model has to consider different cost factors for a single task. Likewise, the infrastructure model has to analyze the layers of the systems stack that comprise a node. The implementation of the system dictates the infrastructure behavior, e.g., by introducing shared resources that are prone to contention. However, we will show that once the workload and the infrastructure are modeled, the allocation problem remains the same for the abstract data-oriented system that covers all classes.

2.2. Data Orientation in DBMSs for Multiprocessor Systems

The second category of data-oriented systems considered in this chapter are database management systems for modern multiprocessor systems. Multi-socket-multi-core systems with eight or more sockets and 64 cores are commonly found in data centers. Larger machines exist and the trend points towards even more cores in a single system. However, classic DBMSs face countless challenges with these systems. The software (not being optimized for these systems) contains many critical sections and central data structures that quickly lead to contention and hinder scalability (e.g., Huber and Freytag, 2009; Johnson et al., 2008, 2009; Pandis et al., 2010; Salomie et al., 2011).

As a consequence of the high main memory capacities in today's servers, modern database systems are very often in the position to store their entire data in main memory. Latency and bandwidth of the main memory are the major bottlenecks of such in-memory DBMSs. The significance of these bottlenecks increases with the trend towards tera-scale multiprocessor systems that exhibit non-uniform memory access (NUMA). On NUMA platforms, each multiprocessor has its own local main memory that is accessible by other multiprocessors via a communication network. Database systems running on NUMA platforms face several issues such as the increased latency and the decreased bandwidth when accessing remote main memory.

Our experiments with three NUMA systems of different sizes show that modern database management systems can achieve the highest performance by treating shared-memory NUMA systems like distributed systems. The results in Figure 2.5 show a factor of up to ten for both latency (10x higher) and bandwidth (10x lower) when comparing local to remote memory accesses (the experiment is presented in detail in Section 7.1). Understanding sockets as individual nodes, which are connected by a fast network and equipped with dedicated memory controllers, and designing for memory locality avoids costly remote memory accesses and yields higher throughputs.

2.2.1. Allocation Problem in DBMSs for Multiprocessor Systems

Database management systems on multiprocessor systems can leverage the data-oriented execution principle to account for the character of the machine as distributed system. Relations are partitioned and partitions together with the operations thereon form tasks. Tasks are mapped to multiprocessors of the machine, i.e., the nodes of the data-oriented system. One way of thinking is to see the storage component in the DBMS as the data-oriented system that offers data containers to the query execution component. Data containers are transparently mapped to actual memory by the allocation strategy. Figure 2.6 sketches data processing on a multiprocessor system depicted as a data-oriented system.

The next section will present various approaches to data management on multiprocessor systems with varying degrees of data orientation. It turns out that not all approaches provide the means to enforce locality of data accesses, e.g., some approaches only partition the logical data organization but not the physical data. The allocation strategy is not that significant in these systems because it only affects a small part of the actual processing.

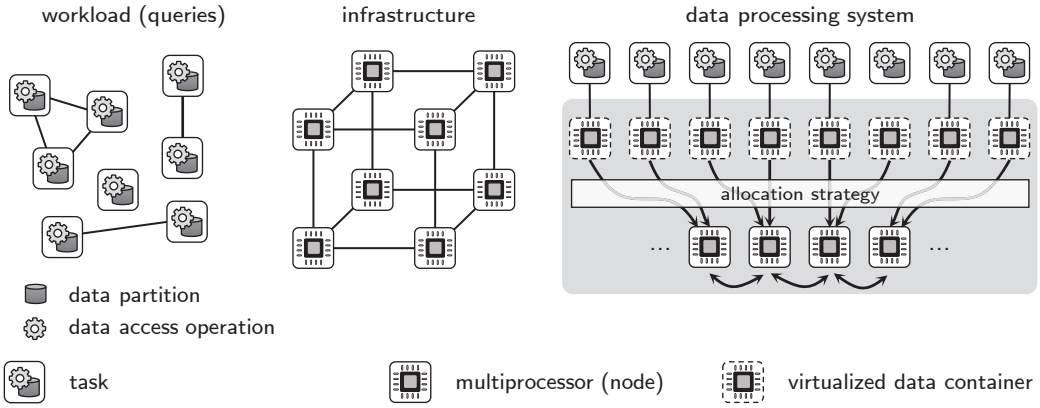


Figure 2.6.: Data Processing on a Multiprocessor System Depicted as Data-Oriented-System

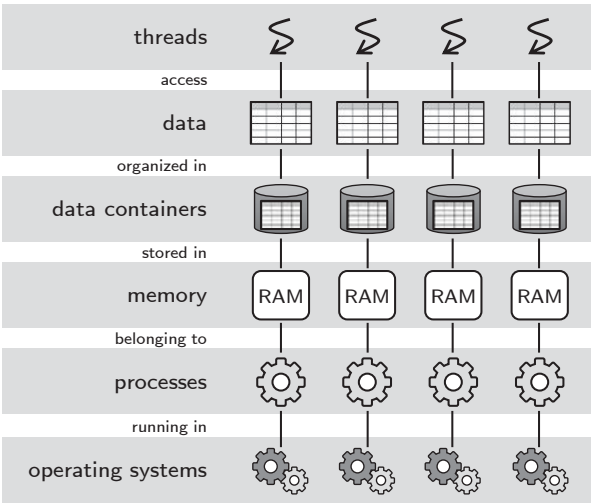


Figure 2.7.: Data Access Stack in Database Management Systems

For systems that provide the means for data locality, solving the allocation problem can be used to achieve various objectives. The throughput can be optimized by balancing load and by co-locating tasks that heavily communicate. The power consumption of the system can be optimized, e.g., in low load phases, by using a smaller number of multiprocessors and sending idle multiprocessors to power-saving modes.

2.2.2. Implementation Classes of DBMSs for Multiprocessor Systems

Different approaches have been proposed to deal with the characteristics of modern hardware in database management systems. To our knowledge, there is no classification of the different systems based on data orientation. We propose a classification based on the abstract data access stack in database management systems shown in Figure 2.7. The classification, shown in Figure 2.8, distinguishes classes based on

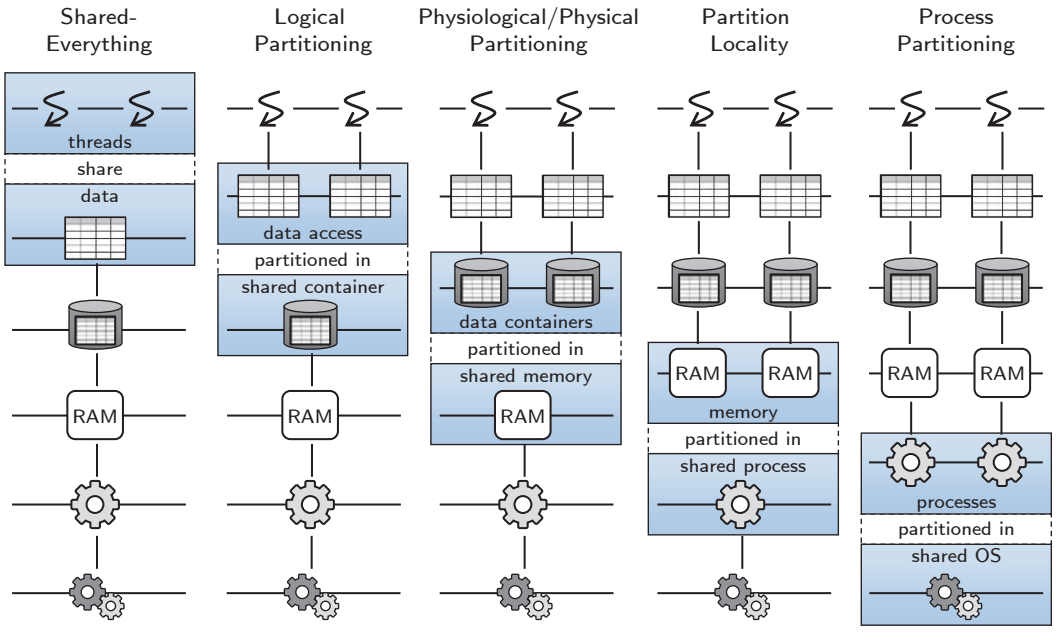


Figure 2.8.: Classification of Data-Oriented Approaches to Modern Hardware

the step in the data access stack, where the data or the access is partitioned. For instance, in the second class (Logical Partitioning), the data, more specifically the logical data access, is partitioned. This classification scheme leads to the following five classes:

1. Shared-Everything (Figure 2.8 on the left),
2. Logical Partitioning,
3. Physiological/Physical Partitioning,
4. Partition Locality, and
5. Process Partitioning (Figure 2.8 on the right).

Details about all five classes are presented in the following paragraphs.

Shared-Everything

The first class, shared-everything, is not actually data-oriented. However, since there are systems that try to tackle the challenges of modern hardware without data orientation, they are shown for the sake of completeness. In shared-everything systems, parallel transactions (threads) access all data concurrently. The data is not partitioned and accesses are not separated. An example for a shared-everything system is the Shore-MT storage manager by Johnson et al. (2009), a multithreaded advancement of the Shore storage manager. Shore-MT improves scalability by optimizing locks, latches, and synchronization (critical sections) and thereby reducing contention on storage manager components like the bufferpool manager or the free space and transaction management. Shore-MT does not consider partitioning the data or the access to the data, the memory layout, or memory locality.

The HyPer system by Kemper and Neumann (2011) is a shared-everything in-memory system written from scratch that uses processor-inherent lazy copy-on-write and virtual memory management to support OLTP and parallel OLAP transactions. Multiple read-only queries can be executed at a time, but writing requests are executed sequentially to ensure consistency without synchronization. This approach obviates the need for locking and latching of data objects or index structures, an otherwise constant source for contention and bottlenecks. HyPer also considers partitions of data to increase the possible degree of parallelism. Partition allows for one writer and multiple readers per partition as well as multiple readers (but no writers) in a shared data partition. However, updating transactions that span multiple partitions require exclusive access to the system like in the initial sequential approach.

Logical Partitioning

Systems that use logical partitioning are the second class of systems in our classification, shown second from the left in Figure 2.8. To our knowledge, Pandis et al. (2010) were the first to use the term data-oriented architecture (DORA) in the context of database management systems (we use the same term but understand it more generally and for a wider class of systems). The authors argue that the execution of transaction processing contains a high number of critical sections and that database management systems face significant performance and scalability problems on highly-parallel hardware. They further argue that the primary cause of the contention problem are the uncoordinated data accesses that are characteristic for conventional transaction processing systems. Based on these observations they propose to couple each thread with a disjoint subset of the database (thread-to-data assignment, or task in our terminology) instead of the classic coupling of a thread with a transaction. As a consequence, each thread can use a light-weight thread-local locking mechanism and contention on global locking structures is severely reduced. Another consequence of the thread-to-data assignment is that load needs to be balanced whenever there is skew in the data or the accesses. Since data is only logically partitioned, i.e., only the access to the data is partitioned and assigned to different threads but the data remains without physical partitioning or alignment, changing the partitioning is a light-weight operation. The authors describe the load balancing mechanism briefly but omit details on the decision making process that leads to balancing actions. Keeping the data physically unpartitioned does not prevent contention on page latches or false page sharing.

Physiological/Physical Partitioning

In a follow-up paper, Pandis et al. (2011) recognize the problem of the overhead and complexity of page latching protocols as well as of conflicts due to false sharing. As a solution, the authors propose the use of multi-rooted B+Trees as a way to enhance logical partitioning and capture most types of physical data accesses as well. This data organization scheme is called physiological partitioning (PLP) by the authors and systems that follow this approach form the third class in our classification in Figure 2.8. Under PLP, a thread is assigned to a sub-tree root of the multi-rooted

B+Tree and it is ensured that requests distributed to each thread reference only the corresponding sub-tree. As a result, threads can bypass the partition mapping and their accesses to the subtree are entirely latch-free. At the same time, the underlying multi-rooted B+Tree supports fast repartitioning and does not require distributed transactions when requests span partitions. Together with the partitioning of the access structures (B+Trees), the authors discuss different partitioning strategies for the underlying heap data. The different partitioning strategies lead to different characteristics with respect to access synchronization and balancing costs.

In contrast to the following class, the various partitions in the PLP system are spread out in memory, agnostic to the NUMA characteristics of multiprocessor systems.

Partition Locality

The fourth class in Figure 2.8, Partition Locality, adds memory partitioning and partition locality to the already partitioned data containers. The ATraPos system by Porobic et al. (2014) uses logical and physical partitioning and relies on data partitioning and placement to maximize locality of data accesses. Adaptive repartitioning is used to maintain data locality under changing workloads. ATraPos furthermore keeps the system state (e.g., locks and lists of transactions) in hardware-aware data structures that require only socket-local data accesses.

The ERIS system (Kissinger et al., 2014) is an in-memory database management system written from scratch that also belongs in this class. ERIS applies many of the aforementioned principles, e.g., logical partitioning, physical data partitioning, and socket-local data structures. We use ERIS for the experimental evaluation of our allocation strategies and will hence describe it in detail in the corresponding Chapter 7.

Leis et al. (2014) present a NUMA-aware query evaluation framework for the previously mentioned HyPer system. The framework is based on the notion of a *morsel*, a partition of data. The size of a morsel is a tuning parameter, experimentally determined to be 100,000 tuples in the paper. Tasks, i.e., units of a pipeline job and a particular morsel to execute on, are distributed at runtime based on utilization and locality (favoring locality but allowing work-stealing).

Process Partitioning

Naturally, classic shared-nothing distributed database management systems can also be deployed on multiprocessor systems. Multiple processes with separate address spaces simulate the various nodes of the network and the multiprocessor interconnects are used for explicit communication. Process Partitioning systems form the fifth class in Figure 2.8.

Distributed shared-nothing databases have been research topics as early as in the 1980s. It is beyond the scope of this thesis to provide a comprehensive overview of all the results. Dewitt and Gray (1992) make a case for shared-nothing architectures and against database machines back in 1992. The article explains many concepts of parallel databases that are still valid today, e.g., scaleup, speedup, data and processing skew, pipeline and partition parallelism, range and hash partitioning. The

authors furthermore give an overview of back then state-of-the-art shared nothing database management systems like Teradata, Tandem, Bubba, and Gamma. Despite using the same principles, the emphases and bottlenecks were different when I/O was dominating, main memory was comparably small and networks slow. For instance, the article states that loading a terabyte database would take 12 days and nights at a megabyte-per-second speed. Apers et al. (1992) even propose PRISMA/DB, a distributed main-memory database. Although having a prototype with 100 processing nodes and a combined main memory of 1.6 GiB, the idea was certainly ahead of its time.

Stonebraker et al. (2007) revisit the idea of shared-nothing database architectures in the context of grids of multi-core systems. The proposed H-Store system deploys an independent instance (i.e., with own indexes, tuple storage, and a disjoint partition of the main memory) on each core of the system. The data is partitioned horizontally and transactions are executed single-threaded and sequentially on each core. Hence, there is no synchronization of access like locks or latches to a data partition needed. However, depending on the data that is accessed, multiple partitions need to communicate and coordinate distributed transactions.

As multi-socket machines with NUMA characteristics gained popularity, researchers played with shared-nothing setups to circumvent remote memory accesses. In Porobic et al. (2012), the authors perform a detailed analysis of different shared-nothing (and shared) deployments in NUMA systems. To show the performance impact of the NUMA architecture on a DBMS, they use multiple instances of ShoreMT. We have previously investigated memory access characteristics and cache effects in NUMA systems based on a synthetic benchmark that mimics a database system's behavior as well as a shared-nothing setup using the MySQL database system (Kiefer et al., 2013).

2.2.3. Summary and Discussion

We showed in this section that many approaches to data management systems on modern multiprocessor systems can be understood as data-oriented systems. We attempted to provide a concise classification for the variety of research prototypes and implemented systems. The classification is based on the step of the access stack where the data access is partitioned and concurrent accesses are isolated. The classification does not fit all systems perfectly, but gives an orientation how the various systems behave. Given the allocation problem, which is the core of this thesis, only systems that provide means for data locality, i.e., systems from the two rightmost classes in Figure 2.8, are good candidates for further investigation. Only when tasks can clearly be separated and assigned to nodes, can the allocation strategy be used to optimize for a given objective. From our experiences and our experiments on NUMA systems, it seems that database management systems that focus on data locality will outperform systems that ignore the physical data layout.

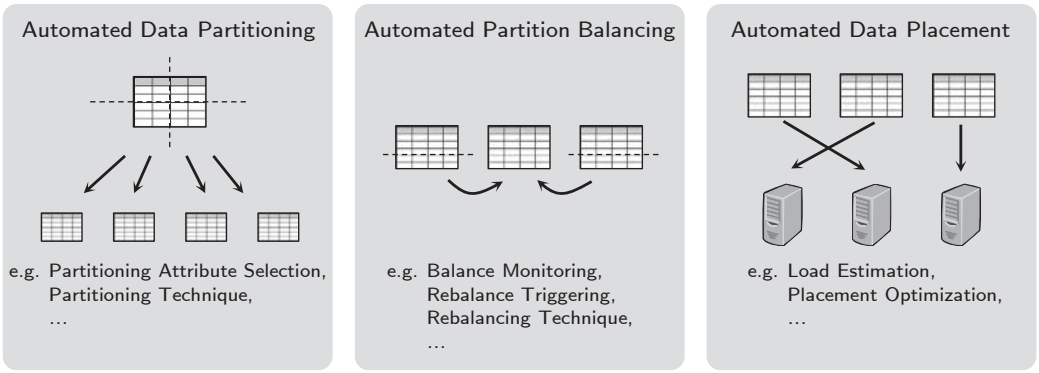


Figure 2.9.: Physical Design Automation Overview

2.3. Physical Design Automation for Data-Oriented Systems

The physical design of a database traditionally comprises several aspects of its actual setup like the creation of indexes or materialized views. In the context of distributed databases, the partitioning of relations and the placement of partitions, hence the allocation, also belong to the physical design. The physical database design greatly influences application performance as it is commonly a tradeoff between performance and storage or maintenance overhead. Also, favoring certain queries in the expected workload over others is a tradeoff to be made by the physical design. Hence, automated or assisted tuning of the physical database design has been a vast research area for decades.

Classic advisors recommend indexes and materialized view (e.g., Agrawal et al., 2000; Zilio, 1997). These advisors frequently use the database system's query optimizer and the underlying cost model in a so-called *what-if mode* where query execution costs are estimated without the need to physically deploy the setup. Together with an algorithm that searches the solution space, the optimizer's estimations can help to find beneficial indexes and materialized views. Index advisors are used at design time (or in certain intervals) and need the expected query workload as input. Like all workload-based design advisors, index advisors assume that the expected workload is known or that a trace can be collected that resembles the actual workload well enough.

Regarding distributed databases, two aspects of the physical design are especially important. First, *partitioning* decides how to split up data. In the field of automated data partitioning, we further distinguish between *partition advisors* that are used at design time with an expected workload and *partition balancers* which react to the actual workload at runtime and try to adapt the partitioning accordingly. Second, *placement* decides where to put each partition and consequently the partitions that are co-located.

Many researchers investigate both aspects, partitioning and placement, but focus on one in particular. Some articles that investigate partitioning also cover placement, often with simplifying assumptions like: (1) always spread data to all nodes, (2) load from different partitions adds up linearly, (3) nodes are all equal, or (4) the network is homogeneous and fully connected. Other works assume the partitioning to be

fixed and only investigate the placement. We point out related work that focuses on the aspects of partitioning and partition placement in the subsequent sections. Figure 2.9 shows the different problems addressed by physical design automation in distributed database management systems. Furthermore, the figure highlights some challenges in each area.

2.3.1. Automated Data Partitioning

The framework of using the query optimizer's cost estimates without physically deploying the setup has been extended to propose data partitioning schemes as well.

Zilio et al. (1994) use a simple heuristic to select partitioning keys: attributes that have the highest weight based on operation type and frequency in the workload (e.g., a join has a higher weight than a group-by). An extension of the algorithm considers the top ranked partitioning key candidates and exhaustively probes all combinations against the optimizer to find the best partition keys. The first author of the paper later substantially extends the algorithms and also considers concurrent reorganization in his thesis (Zilio, 1997).

First to solely use the cost model of the optimizer, as opposed to heuristics or a separate cost model, were Rao et al. (2002). Based on a given workload of SQL statements, the authors seek to determine how to partition the base data across multiple nodes in a shared-nothing parallel database management system. A rank-based enumeration method is used to find candidate solutions and so-called *interesting partitionings* are considered, i.e., partitionings that are not optimal for certain queries but benefit the global cost. Replication of tables is not considered by the authors.

Agrawal et al. (2004) extended the idea to incorporate vertical and horizontal partitioning into automated physical database design. The authors propose an integrated solution to partitioning, indexes, and materialized views again based on the expected SQL workload and costs estimated by the optimizer. A similar approach was proposed by Zilio et al. (2004), a continuation of the author's previous work.

A more recent approach by Nehme and Bruno (2011) considers partitioning and replication of tables. Given a database, a query workload, and a storage bound, the authors' method tries to find a partitioning strategy such that the size of replicated tables does not exceed the storage bounds and the overall cost for the workload is minimized. Unlike previous approaches, this algorithm is deeply integrated in the query optimizer to improve the search for candidate solutions, e.g., by early pruning of partial configurations.

A slightly different approach by Pavlo et al. (2012) is based on a separate analytical cost model and large neighborhood search to explore the solution space. The authors try to minimize distributed transactions in the shared-nothing system H-Store while mitigating effects of temporal skew. The developed tool, Horticulture, considers horizontal partitioning, replication, replicated secondary indexes, and stored procedure routing. The authors specifically target OLTP workload and make strong assumptions on the database schema and workload, e.g., a hierarchical schema.

A project called Schism by Curino et al. (2010) aims at minimizing the number of distributed transactions by assigning each tuple individually to a partition such that the number of transactions that span multiple partitions is minimized. The

authors use graph partitioning techniques to come up with an assignment of tuples to partitions based on the workload, which is modeled as a graph. The idea of modeling the workload as a graph and using graph partitioning algorithms to find a partitioning is picked up by Kumar et al. (2013, 2014) in the SWORD project. Both Schism and SWORD will be revisited in Section 3.5 where we present an in-depth analysis of closely related approaches to the allocation problem for data-oriented architectures.

2.3.2. Automated Partition Balancing

Partition balancers are based on the simplifying assumption that balancing load across all partitions leads to the best physical setup. Unlike partitioning advisors, balancers try to improve the partitioning at runtime. Partition balancers can be used with any initial partitioning to eventually find a balanced partitioning. Furthermore, partition balancers can naturally react to changing workloads. The main challenges are (1) to monitor the system behavior with minimal impact on performance to be able to detect imbalances and (2) to quickly react to imbalances with (ideally incremental) new partitioning rules.

Early work on load balancing in shared-nothing database systems, e.g., by Rahm and Marek (1993), assumes that in parallel join processing, relations would by dynamically redistributed to the actual join processors. Based on the current CPU utilization and a cost model, the number of join processors as well as the active join processors are selected dynamically.

Scheuermann et al. (1998) investigate load balancing in shared memory parallel disk systems. The authors introduce a notion of *disk heat* based on inter-arrival times of requests. An online greedy heuristic tries to dynamically balance load (referred to as *disk cooling*) by migrating data away from the hottest disks. The migration mechanism is invoked at fixed intervals and only if certain conditions are met.

The abovementioned ATraPos system (Porobic et al., 2014) dynamically repartitions data to balance resource utilization while minimizing transaction synchronization overhead. Based on monitored metrics, e.g., costs for actions on subpartitions, and a cost model, an iterative algorithm tries to improve the partitioning by moving work to the most under-utilized core. ATraPos uses thread-local monitoring data structures and eventual global aggregation to reduce the impact of the monitoring on performance. Once a set of repartition actions, i.e., split or merge, is found, the regular executions is paused to migrate the data. The balancing algorithm is invoked at flexible intervals, depending on the characteristics of the workload.

The morsel-driven query evaluation framework introduced before (Leis et al., 2014) does not balance the partitioning in the classic sense by invoking a separate balancing algorithm. Instead, each worker is able to steal work from other sockets in the absence of local tasks. By writing results to local memory, a worker that steals work implicitly changes the partitioning of the data. A lock-free meta data structure, i.e., a list of available and executable tasks (referred to as *dispatcher* in the paper), is maintained and used by workers to determine the next task to execute and hence the next data partition to work on.

2.3.3. Automated Data Placement

Besides data partitioning, data placement is the second big challenge in distributed databases. Once data is split into a given number of partitions, each of the partitions needs to be assigned to a node in the system. A simplification of the problem is to assume equal nodes and a homogeneous topology. Then, the problem is reduced to building as many subsets of partitions as there are nodes.

Mehta and Dewitt (1997) analyze data placement in shared-nothing parallel database systems. The authors state that in the absence of remote access, the data placement determines the distribution of data and the corresponding operators. The paper considers declustering to balance load across the nodes of the shared-nothing system. A method, based on simulations of the DBMS as closed queuing system, is presented to determine the degree of declustering and the subset of nodes to store the data. The quality of the method depends on the ability to model the DBMS and the underlying hardware and the accuracy of the simulator.

Schaffner et al. (2013) propose the Robust Tenant Placement (RTP) problem that assigns tenants to nodes in a multi-tenancy database-as-a-service system. The RTP and the methods used to solve it will be revisited in Section 3.5.

Curino et al., 2011 investigate workload-aware database consolidation. The presented Kairos system consolidates databases based on the predicted combined resource utilization. The Kairos system will also be revisited in Section 3.5.

2.4. Summary

In this chapter, we presented foundations of data-oriented systems. We introduced two application scenarios, *database-as-a-service* and *DBMSs for multiprocessor systems*, that use data-oriented systems. Classifications of systems, based on the extent of data locality and the proportion of private and shared resources, were proposed in both scenarios. Different characteristics and tradeoffs of the various different implementations lead to valid use-cases for all classes. Depending on the implementation, the workload model has to consider different cost factors for a single task. Likewise, the infrastructure model has to analyze the layers of the systems stack that comprise a node. We will show in the next chapter that all systems can use a unified allocation strategy once the workload and the infrastructure of the data-oriented systems are abstracted in the corresponding models.

Second in this chapter, we recapitulated related work in the field of physical design automation in distributed database management systems. Approaches to automatic data partitioning, partition balancing, and partition placement are important foundations to better understand the allocation problem in data-oriented systems.

In the next chapter, we present the allocation problem for data-oriented systems. We first abstract from specific data-oriented systems to generic infrastructures and workloads and then formulate the actual allocation problem.

3. Allocation Problem for Data-Oriented Systems

In this chapter, we introduce the allocation problem for data-oriented systems. We showed in the previous chapter that a wide variety of actual systems with different characteristics implement data-oriented architectures. The allocation strategy to solve the allocation problem in this thesis is applicable to data-oriented systems in general. However, to use the allocation strategy, specific data-oriented systems have to be generalized to an abstract data-oriented system. The abstract system is a common denominator of all data-oriented systems and acts as an interface to the actual allocation strategy. Figure 3.1 sketches the generalization of a (class of) database-as-a-service system(s) to the abstract data-oriented system. The hardware resources of the system are collected in the *infrastructure*. The infrastructure model, detailed in Section 3.1, describes system abstractions and necessary information to derive from the actual system. The applications that run in the database-as-a-service system are collected in the *workload*. The workload model in Section 3.2 details how to describe a workload and provides hints on how to obtain the workload. Since actual data-oriented systems differ greatly, methods to obtain workload and infrastructure information for a given system have to be provided by domain experts.

Given the infrastructure and workload information, the allocation problem itself is formalized and variations thereof are discussed in Section 3.3. In general, the allocation problem is to find a mapping from workload tasks to infrastructure nodes. Assuming that there are considerably more tasks than nodes, several tasks share nodes and the mapping determines the overall system performance as well as other system properties. Consequently, the allocation strategy, which provides that mapping, is used to optimize the system. Different objectives, e.g., performance or resource consumption, and corresponding constraints, e.g., given resources or performance guarantees but also availability, maintainability, or isolation, are discussed in Section 3.3.

In the simplest case, the workload is static and the infrastructure is fixed. In this case, the allocation strategy can be used to find a mapping at compile time (offline) that is once implemented and never changed. When the workload or the infrastructure change dynamically, the allocation strategy has to adapt the mapping at runtime (online) to compensate for the changes. In this thesis, we assume dynamic workloads and infrastructures[1] and our allocation strategy is able to incrementally update a mapping.

[1] In our models, workload and infrastructure are constant for (possibly short) intervals of time and may change between these intervals. This approach differs from, e.g., modeling the workload as a continuous time series.

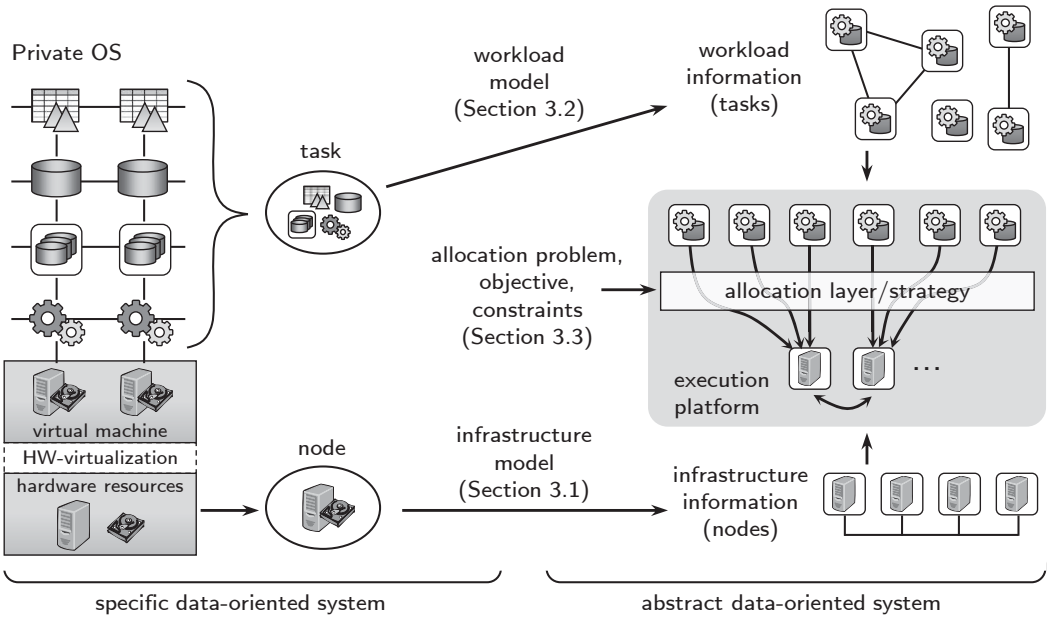


Figure 3.1.: Abstracting a Specific Data-Oriented System

Data Partitioning Problem The allocation problem takes workload tasks and assigns them to infrastructure nodes. Another problem, the *data partitioning problem*, is closely related to the allocation problem and sometimes the two problems are treated together. The data partitioning problem is to decide whether or not and into how many pieces to partition a database object. Depending on the specific data-oriented system, this may include the selection of partition keys, key ranges, or individual tuple-to-partition mappings. In this thesis, we mostly assume that the data partitioning problem is solved separately before the workload is abstracted in the workload model. However, we briefly discuss the data partitioning problem here and again later in this chapter to highlight the correlation to and the interactions with the allocation problem.

Both, the data partitioning problem and the allocation problem are loosely connected because decisions made in solving one problem may influence the solution of the other problem. If, for instance, a relation is partitioned into many small pieces, each piece may fit on a processing node that is too small to hold the entire partition. Therefore, the solution space for the allocation problem changes. The other way around, assigning a relation to a node with very limited scan performance may encourage the partitioning algorithm to split the relation and scan it in parallel.

The data partitioning problem has been of interest ever since the first partitioned databases were available (e.g., Ries and Epstein, 1978). Aspects to consider in the solution are the size of the relations and the capacities of the nodes, i.e., relations that do not fit on a single node always have to be partitioned. When partitioning is optional, it is a tradeoff between costs induced by communication and distributed protocol overhead and gains caused by the use of additional resources. Depending on the workload it may be beneficial to decluster, i.e., spread out, a relation to utilize the combined bandwidth of several nodes in a large scan operation. When relations

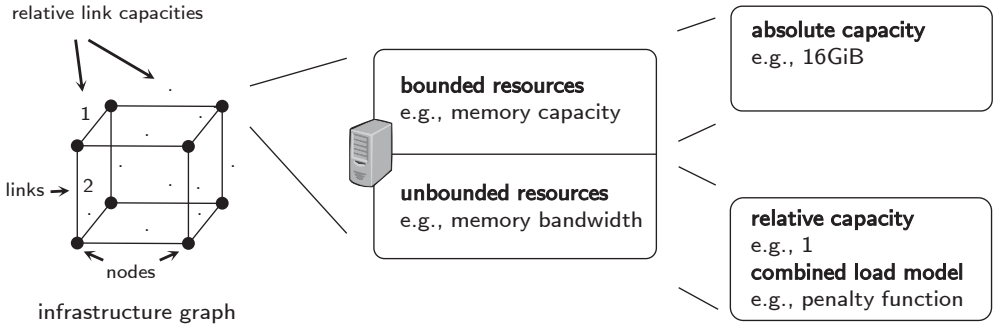


Figure 3.2.: Infrastructure Components

are joined in the workload, selecting the appropriate partition keys is crucial to enable the allocation algorithm to co-locate partitions of relations based on join keys to avoid costly communication.

As stated previously, we focus on the allocation problem, i.e., we compute an initial partitioning based on simple heuristics or assume the existence of initial partitions. For instance, existing partitioning advisors can be used to provide an initial partitioning. However, we will point out how the partitioning impacts the allocation and vice versa and we will describe basic strategies to incrementally update the partitions.

Chapter Outline The infrastructure model and the workload model are detailed in Sections 3.1 and 3.2 respectively. In Section 3.3, the allocation problem is formalized and possible objectives and constraints are discussed. From the problem description, we derive and detail requirements for allocation strategies in Section 3.4. Last, in Section 3.5, we discuss related approaches proposed in the literature for variations of the allocation problem.

3.1. Infrastructure Model

In this section, we describe the model that abstracts from the actual hardware of the execution environment to a generalized *infrastructure*. Characteristics of the execution environment that are needed to solve the allocation problem are subsumed in the infrastructure. The infrastructure model contains the hardware resources that are captured in the infrastructure (Section 3.1.1) as well as their types (Section 3.1.2) and their models for combining loads (Section 3.1.3).

Definition 3.1 (Infrastructure)**.** An *infrastructure* is an undirected graph of *nodes* connected by *links*. Nodes have bounded and unbounded resources and weight functions map each node to either an absolute or a relative capacity per resource. In the case of equal nodes, the relative capacities for unbounded resources can be omitted. Each unbounded resource additionally has a model for combining loads. Link capacities in the infrastructure are given by an edge-weight function. To solve the allocation problem, an infrastructure must also provide a routing function that maps

any two nodes to a set of links that connect the two nodes. For fully connected networks, the routing function can be omitted.

All components of an infrastructure are visualized in Figure 3.2. Although not explicitly stated, the infrastructure definition implies that the infrastructure can be heterogeneous with respect to the nodes and the links. Nodes and links in the infrastructure can have different capacities. Additionally, the network can be sparse, i.e., nodes may not be fully connected.

The granularity of a node in the infrastructure depends on the implementation of the data-oriented system, e.g., it may be a single core or all combined multiprocessors in a system. Likewise, a link may refer to an internal link between multiprocessors or a network connection between servers.

3.1.1. Modeled Resources

The infrastructure in Definition 3.1 can be used to model different kinds of actual hardware resources. The allocation strategy finds mappings based on the abstract infrastructure instead of any specific resource. To illustrate the infrastructure, we exemplarily describe three main resources:

Processing Resources: CPUs in each node are abstracted to processing resources. Processing resources can, e.g., be characterized by and measured in instruction throughput, i.e., retired instructions per second. Depending on the actual system and the granularity of the data allocation, processing resources can refer to anything from a single core in a multi-core processor to all combined multiprocessors in a system.

Memory Resources: Main memory attached to a node is abstracted to a memory resource. Again, the memory resources can abstract a single memory controller or several memory controllers based on the granularity of a node in the infrastructure. Memory can, e.g., be characterized by a capacity and an access bandwidth.

Network Resources: The network that connects nodes is also modeled in the infrastructure. The network can be characterized by link bandwidths.

Since the actual data-oriented systems that we use in this thesis are based on in-memory database management systems, we omit disks and disk I/O in the infrastructure. However, the infrastructure definition does not explicitly exclude I/O, which can be modeled as a resource if needed.

3.1.2. Bounded and Unbounded Resources

The infrastructure model distinguishes between *bounded resources* and *unbounded resources*.

Bounded Resources have a hard limit that cannot be exceeded. Bounded resources cannot be overcommited and overloading a node's bounded resources leads to

invalid allocations. The infrastructure must provide capacities, i.e., upper bounds, for all bounded resources. For homogeneous nodes, a single global upper bound is sufficient. For heterogeneous nodes, the upper bound has to be provided per node.

Unbounded Resources are not literally unbounded but unlike bounded resources they can (and usually will) be overcommited. Overcommitting an unbounded resource still leads to a valid allocation, even if the performance degrades. The infrastructure does not contain upper bounds for unbounded resources. However, relative capacities need to be provided in the case of a heterogeneous nodes to account for individual nodes' capacities.

We assume that bounded resources show constant behavior up to the upper bound. For instance, we assume that access performance does not depend on the size of the data in memory. Unbounded resources on the other hand show degrading performance with increasing load and overcommitment (with a possibly non-linear dependency). The goal of our allocation strategy will be to respect the upper bounds of bounded resources and to balance unbounded resources. The intuition behind this strategy will be discussed in Section 3.3.

A given hardware resource rarely is strictly a bounded or an unbounded resource. Each resource considered in the previous section can equally be modeled as a bounded or an unbounded resource and the decision affects the workload model, the allocation strategy, and the resulting allocation. For instance, the main memory capacity can either be considered a bounded resource or an unbounded resource in the presence of paging mechanisms. However, the expensive I/O operations caused by paging decline performance drastically and it may be better to model the main memory capacity as a bounded resource. Consequently, any allocation that overloads a single node's main memory is considered invalid. In contrast, we model processing resources, memory bandwidth, and the network as unbounded resources where overcommitment still leads to valid allocations. All these resources can also be modeled as bounded resources. Taking the example of CPU instructions, there is an upper bound for the number of instructions that a given CPU can retire per unit time. If the number of CPU instructions is known for the workloads, modeling CPU instructions as a bounded resource can lead to tight performance guarantees where either each workload is able to retire a given number of CPU instructions or (if not) the allocation of workloads to the CPU is considered invalid and rejected. Modeling resources as being bounded or unbounded depends on the intention of the model and the availability of resource capacity and workload requirement information. Bounded resources only require relative capacities, which are usually easier to obtain.

3.1.3. Models for Combined Load

Several tasks commonly share a single node in data-oriented systems. To be able to evaluate a given allocation, the global load of a node, i.e., the combined load induced by all tasks that are executed on the node, needs to be estimated. A node's load determines it's performance and is hence used to achieve the objective

of the allocation problem and to formulate constraints of the allocation strategy (the correlation of load and performance is discussed in Section 3.3).

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, in practice, performance often depends on all kinds of workload parameters like request rates, request types, request sizes, and the concurrent execution of requests. Contention on resources caused by concurrent execution may lead to performance the does not scale linearly with the amount of work. Therefore, in addition to the *linear* model, we propose a *non-linear* model to combine the individual loads induced by tasks that share a node.

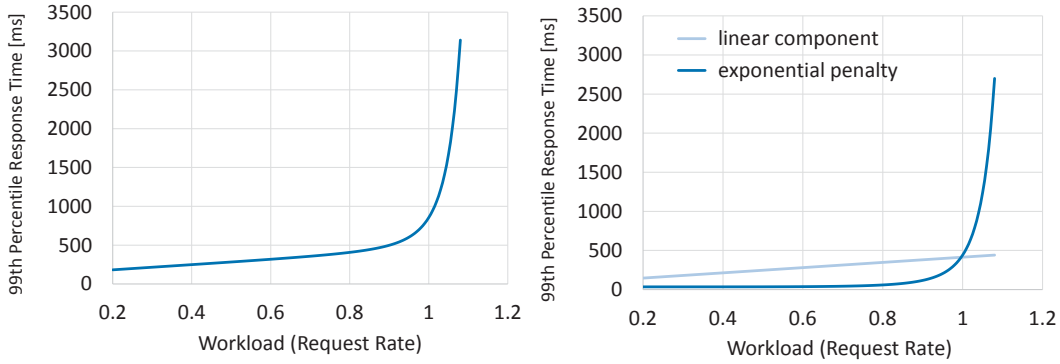
Note that this section discusses how loads, i.e., tasks, are combined. Consequently, this discussion could arguably also be part of the workload model in the following section. In fact, the later allocation strategy will apply the non-linear model to the workload (by increasing the load induced by all tasks) in contrast to applying it to the infrastructure (by reducing a node's capacity depending on the tasks). However, since the non-linear performance is caused by contention in the infrastructure, we consider it a property of the same and decided to discuss it here.

We casually refer to resources that use non-linear models to combine loads as *non-linear resources* and in general we refer to *non-linear resource usage* and *non-linear resource behavior*. When using these short (though not entirely correct) terms, we refer to the non-linear performance (non-linear in the amount of work) that results from using the resources, e.g., caused by contention.

Non-Linear Resource Usage

A major reason for sub-linear scalability with the number of tasks is contention on resources. Contention can happen on the hardware itself, on operation system resources, or on DBMS components. Disk I/O, as a prime example, has always been considered a non-linear resource caused by characteristics of the underlying hardware. With our focus on in-memory technology, disk I/O is beyond the scope of this thesis. However, other resources like CPU and memory that are sometimes considered to be linear resources for simplification also show non-linear behavior under high load.

Depending on the implementation, high request rates can lead to a high number of context switches in the operating system (process or thread switches). Li et al. (2007) measure direct costs of context switches, e.g., caused by saving and restoring registers and by system calls, and indirect costs of context switches caused by cache pollution. With a tool provided and discussed by Sigoure (2010) in a blog post, we measured the direct costs of a context switch on a current Intel Xeon E7 to be about 32 µs. With a high degree of parallelism, this can add up and deteriorate performance. A high request rate also causes contention on other resources. We have investigated the effect of cache sharing between database workloads in a NUMA system in a previous article (Kiefer et al., 2013). Tözün et al. (2013) investigate cache misses that occur when executing an OLTP system. The authors break down the cache misses and identify components of the system that cause certain types



(a) Experimental Response Time Based on Request Rate

(b) Decomposition of Response Time Function

Figure 3.3.: Non-Linear Response Time Behavior, from Schaffner et al. (2011)

of cache misses. Blagodurov et al. (2010) conduct detailed experiments on memory controller contention, memory bus contention, and prefetching hardware contention.

The data-oriented system itself is another source for contention. As shown in Section 2.1.2 for database-as-a-service systems, the layers of the system stack that belong to the node and are hence captured in the infrastructure model depend on the implementation of the system. For instance, given the Private Database class, the database server is part of the node and contention on DBMS resources is captured in the non-linear behavior of the infrastructure. Different components of relational database management systems like logging facilities, free space management or transaction management can congest under high load. Johnson et al. (2008) investigate contention on various components of the DBMS that lead to bad scalability. In a related work (Johnson et al., 2009), the authors analyze the storage manager Shore and identify several components, e.g., buffer pool manager, log manager, and lock manager, that cause bad scalability. The optimized Shore-MT improves scalability by optimizing locks, latches, and critical sections. Pandis et al. (2010) use bad scalability caused by contention on DBMS resources as motivation for their data-oriented architecture. Although data-oriented systems aim at localizing work and avoiding central data structures, there always remains a portion of central code that may cause contention.

In summary, the detailed behavior of the system strongly depends on details of the hardware, implementation details of the operating system, and on the data-oriented system itself. To grasp the general behavior of the complex system, we assume a simplified resource consumption model that reflects the non-linear performance of the system. In our infrastructure model, each resource has a (relative) capacity and a (possibly non-linear) model to combine loads. We distinguish two types of non-linear behavior, (1) penalized resource usage and (2) general non-linear resource usage. The first type, which is a special case of the second type, is intuitive and leads to optimizations in the allocation strategy.

Penalized Resource Usage The penalized resource usage model is a combination of the linear model and a non-linear penalty function. Up to a certain load or degree

of parallelism, the linear usage assumption often holds and the performance can be predicted accurately. However, when a certain load level is reached, contention occurs and the performance does not scale linearly beyond this load level. 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.

Schaffner et al. (2011) did an experiment with the in-memory column database TREX. The authors increase the abstract workload, a combination of data size and request rate, and measure the 99th percentile value of the response time of all requests. Based on multiple experiments, the response time is approximated as a function of the workload. The results of the experiment are shown in Figure 3.3. Figure 3.3a shows the results from the paper for a fixed data size, i.e., when the workload is only a function of the request rate. Figure 3.3b shows that the function for the response time can be split into a linear component and a penalty that starts to affect the response time above a certain load level. The authors in the paper avoid load above this point and argue in favor of a linear performance model. We assume overcommited resources and argue in favor of incorporating the penalty in the infrastructure model.

General Non-Linear Resource Usage In the general case, non-linear resource usage can be modeled as an arbitrary function[2] of one or more linear resources. In contrast to penalized resources usage, the non-linear function here does not have to be a composition of the linearly combined load and a penalty. Furthermore, multiple resources together can be used to describe and thereby influence a single non-linear resource.

Note that the non-linear usage model of resources in the infrastructure model is still a simplification of the actual system behavior. However, given the various sources for contention, we consider it more accurate than the simpler linear model. Given the dependency of the resource behavior on the underlying system, we do not detail any particular non-linear model here but assume the existence of penalty functions or general non-linear resource usage functions. In an actual system, these functions have to be determined from manufacturer manuals, operating system documentation, or experimentally based on low-level experiments using the monitoring capabilities of the system.

3.2. Workload Model

In this section, we describe the model that abstracts from the actual tasks to a generalized *workload*. Characteristics of the tasks that need to be available to solve the allocation problem are subsumed in the workload. Workload and infrastructure are similar in structure, i.e., modeled as a graph, to enable later allocation strategies.

Definition 3.2 (Workload)**.** A *workload* is an undirected graph of *data partitions* connected by *data transfers*. Data partitions and operations executed on them are

[2]The non-linear function must be positive and monotonic increasing. This requirement does not impose a strict limitation, as the resource usage usually increases when load is added. The reason for the requirement will be detailed when the allocation strategy is described in Chapter 4.

bundled in *tasks*. Tasks consume bounded and unbounded resources and vertex weight functions map each task to either an absolute or a relative cost, i.e., load, per resource. An edge weight function quantifies data transfer costs.

In this thesis, we exemplarily assume tasks to induce a relative processing cost and a relative memory transfer cost (unbounded resources). Furthermore, data partitions consume an absolute memory capacity (bounded resource). Edge weights in the workload represent amounts of data being transferred between data partitions. The granularity of a data partition in the workload depends on the implementation of the data-oriented system, e.g., it may be a single tuple, a relation or parts thereof, or a whole database. Likewise, a task is a generalization of a query, a transaction, or any complex set of operations on the data.

Note that we use the terms *cost* and *load* interchangeably when referring to a task's activity. Most precisely, a task's *load* refers to the best-case resource consumption of that task. For instance, if a task is executed under ideal conditions (i.e., isolated in a dedicated environment) and is there able to read from memory at a rate of 5 GiB/s, then this is the task's load with respect to the memory controller bandwidth. Recall that for unbounded resources, relative costs are sufficient and usually easier to obtain. For instance, it is sufficient to know that a certain task induces twice as much load as another task.

The abstract workload is a generalization of the work done in the data-oriented system. The workload graph acts as an interface between the actual tasks performed in the system and the allocation strategies. Data-oriented systems may look differently and it heavily depends on the actual system how the workload graph can be derived from the performed tasks. In fact, the workload does not even have to be in the form of SQL statements. A program, written in a high-level language, that contains data manipulating operations may as well be transformed to a workload graph. Related, though in detail different examples of modeling workloads in data-oriented systems can for instance be found in Curino et al. (2010, 2011) and Schaffner et al. (2013).

3.2.1. Obtaining the Workload

In this section, we present an exemplary scenario using the data-oriented in-memory DBMS ERIS (see Chapter 7). We detail the steps and necessary information to transform a list of SQL statements to a workload graph. The steps to obtain the workload graph may be different for other systems.

In this scenario, the workload information is provided as a list of SQL statements and execution frequencies, e.g., obtained from a recorded trace. It is furthermore assumed that the database can be deployed and the statements can be executed in the DBMS to gather execution information and statistics (otherwise, these information have to be provided together with the input). Collecting the necessary information induces minimal overhead and can either be done in a dedicated staging area of the system, where workloads are evaluated before entering the system, or directly in the productive system. Along the way, obtaining the workload graph requires to assign resource usage costs to basic DBMS operations. These costs can either be derived

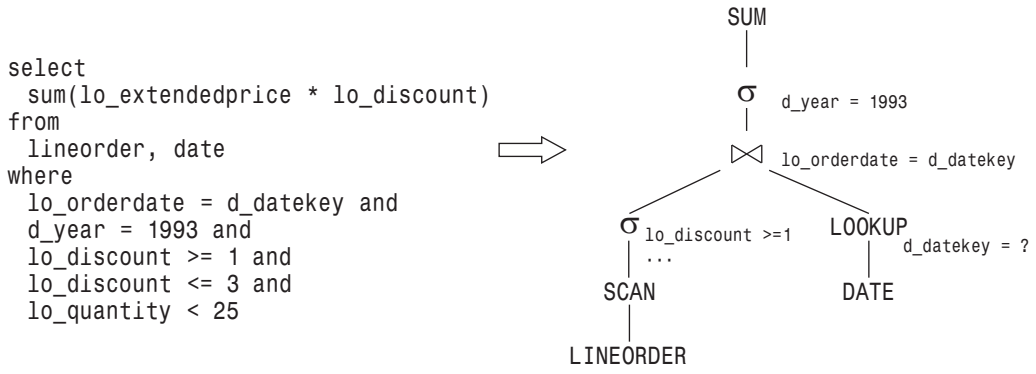


Figure 3.4.: Transforming the SQL Statement to the QEP

from an analytical cost model or from sandbox experiments (or a combination of both).

Given these preconditions, a workload graph is derived from a single SQL statement in three steps. Multiple workload graphs from individual statements are aggregated to the global workload graph by summing up vertex and edge weights. The three steps to obtain a workload graph in ERIS are:

1. Transform the SQL statement to the Query Execution Plan (QEP),
2. Transform the QEP to an intermediate graph representation, and
3. Transform the intermediate graph representation to the workload graph.

All three steps are detailed in the following paragraphs. An example of the intermediate representations is shown in Figures 3.4, 3.5, and 3.6. The example uses query 1.1 from the Star Schema Benchmark (O'Neil et al., 2009).

SQL Statement to QEP In the first step, shown in Figure 3.4, a query execution plan (QEP) is built from the SQL statement. The QEP is a DBMS specific internal representation of the statement that shows several execution details of the statement. The QEP contains all input relations (or partitions of relations) and their access strategies. Additionally, the QEP contains relational operators like joins, aggregations, or selections.

QEP to Intermediate Graph In the second step, shown in Figure 3.5, the QEP is transformed to an intermediate workload graph. To process the QEP, it has to be annotated with cardinality information obtained during a statement execution (or estimates based on statistics). In Figure 3.5, the required information is marked with encircled numbers, i.e., the sizes of the input relations and cardinality information along the execution tree. From the annotated QEP, an intermediate workload graph representation is derived where each relation (or partition) is a vertex. Relations, which are connected by join or union operators, are connected by edges in the graph. In ERIS, the leftmost relation in the QEP drives the execution, i.e, it collects all data from other relations and returns the final result. Vertices in the intermediate graph representation are annotated with sizes and operator lists containing intermediate

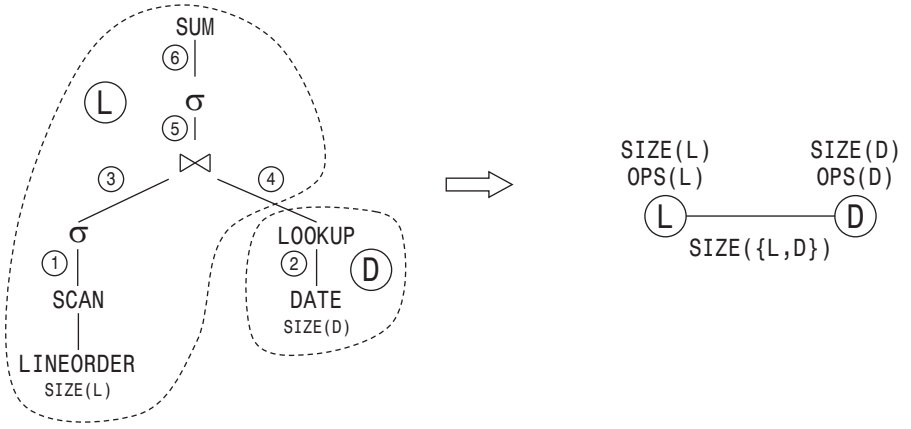


Figure 3.5.: Transforming the Annotated QEP to an Intermediate Workload Graph

cardinalities from the QEP. In the example shown in Figure 3.5, let C_1 be the cardinality of the intermediate result at the position marked with ① (likewise for all intermediate results). It follows that OPS(L) of vertex L is a list of operators that contains scanning C_1 tuples from the `lineorder` relation and applying predicates to C_1 tuples. Furthermore, C_3 keys have to be sent to vertex D for lookup, a predicate has to be applied to C_5 tuples, and the remaining C_6 tuples have to be aggregated in a sum. The operations list OPS(D) of vertex D only contains looking up C_3 keys and returning C_4 matches. The edge {L,D} is annotated with a size that comprises C_3 lookup keys and C_4 tuples from the `date` relation. The graph annotations in the example are:[3]

$$\begin{aligned}
\text{SIZE(L)} &= (\text{TUPLE}(C_1)),\\
\text{SIZE(D)} &= (\text{TUPLE}(C_2)),\\
\text{OPS(L)} &= (\text{SCAN}(C_1), \text{SELECT}(C_1), \text{SEND}(C_3), \text{SELECT}(C_5), \text{SUM}(C_6)),\\
\text{OPS(D)} &= (\text{LOOKUP}(C_3), \text{RETURN}(C_4)),\\
\text{SIZE(\{L,D\})} &= (\text{KEY}(C_3), \text{TUPLE}(C_4)).
\end{aligned}$$

In this simplified notation, TUPLE(C_1) for instance denotes the size of a tuple of the `lineorder` relation multiplied with the cardinality C_1.

Intermediate Graph to Workload In the last step, shown in Figure 3.6, the intermediate workload graph is transformed to the actual workload graph by mapping vertex and edge annotations to resource usage costs. The sizes of the input relations together with intermediate data are bounded resources and are mapped to $S(\cdot)$. The operations lists of the intermediate graph are mapped to processing costs $P(\cdot)$ and memory transfer costs $M(\cdot)$ using a cost model. The cost model can be the internal cost model of the DBMS, a separate cost model that reflects the DBMS, or a

[3]Note that this is a simplified representation for readability purposes. For instance, to derive the actual size of the lineitem relation, not only the cardinality C_1 but also the size of an individual tuple is required. Similarly, other operators require additional information to derive costs.

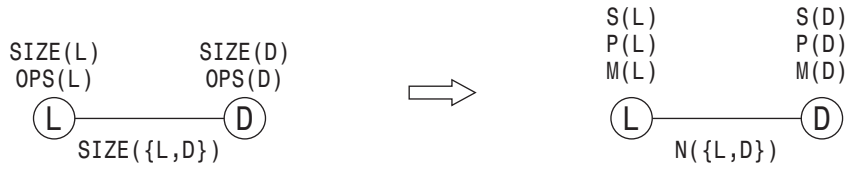


Figure 3.6.: Transforming the Intermediate Workload Graph to the Final Workload Graph

generic cost model (e.g., Manegold and Boncz, 2002). Since processing and memory access are unbounded resources, the corresponding costs can be provided as relative costs. Network cost $N(\cdot)$ are derived from the edge annotation of the intermediate graph. To complete the example, assuming a cost model, the (normalized) costs of the workload may be:

$$\begin{aligned} \mathrm{S(L)} &= 567\,\mathrm{MiB} & \mathrm{P(L)} &= 13.3 & \mathrm{M(L)} &= 48.4 \\ \mathrm{S(D)} &= 225\,\mathrm{KiB} & \mathrm{P(D)} &= 1 & \mathrm{M(D)} &= 1 \\ \mathrm{S(\{L,D\})} &= 94\,\mathrm{MiB}. \end{aligned}$$

The relative costs denote that executing all operators that comprise the vertex L is 13.3 times more expensive with respect to processing compared to executing vertex D. Likewise, vertex L is 48.4 times more expensive than vertex D with respect to memory access.

3.2.2. Maintaining the Workload

We assume the workload to be dynamic in nature. Hence, part of the system and the allocation strategy is to maintain workload information in presence of changing tasks to be able to adapt the allocation accordingly. Maintaining the workload is a twofold problem, (1) detecting the changes and (2) incorporating the changes in the workload graph.

How changes in the workload can be detected strongly depends on the implementation of the data-oriented system. Monitoring facilities must be provided that detect changes of the incoming tasks as well as changes in the data characteristics, e.g., cardinalities or data skew.

Adding and removing tasks can be accomplished by adding or subtracting the corresponding vertex and edge weights. Once detected, a change in the incoming workload may be incorporated in the workload graph in different ways. The vertex and edge weights that belong to the task have to be removed from the global workload. Then, the task's workload graph can either be updated, e.g., when cardinalities have changed, or built from scratch. The modified workload for the changed query can then be added again to the global workload.

When the partitioning of a data object is modified, e.g., when a relation is split or two partitions are merged, the workload graph can be incrementally updated by splitting or merging, respectively, the corresponding vertex. Vertex and edge weights have to be assigned to the newly generated vertex (or vertices) and edges accordingly.

3.3. Allocation Problem

With Definitions 3.1 and 3.2 for an infrastructure and a workload, respectively, the allocation problem can now be discussed and finally formulated.

3.3.1. Possible Objectives

The allocation problem can be formulated differently to optimize for different objectives (see also Section 3.5 with related approaches). Two basic formulations are:

1. Minimize the number of used resources without violating performance constraints!
2. Maximize performance with a given amount of resources!

Both formulations can be extended and customized to further specify the allocation problem. Performance constraints and objectives can for instance refer to different performance metrics like throughput or response time. Furthermore, performance goals can be formulated for average or worst case behavior and different performance levels can be requested to offer different service levels. Resource constraints and objectives can for instance refer to the number of nodes, the network communication, or the total energy consumption (e.g., Dargie et al., 2011).

In this thesis, the objective of the allocation problem is to find a valid mapping from the workload to the infrastructure that optimizes the workload performance with a given and fixed number of nodes. Details about the performance metrics used in our experiments are provided in the corresponding Chapters 6 and 7.

Meta-Objectives Given the complexity of the underlying systems, certain behavior may be hard to evaluate without actually materializing a setup and executing the workload. Especially, predicting the performance of a system with respect to response times and throughput is close to impossible due to the complex interactions of concurrently executed tasks. Instead, meta-objectives that can be evaluated are used in practice to achieve the actual goals. For instance, the Schism project (Curino et al., 2010) uses the number of distributed transactions, i.e., the message count[4], as a meta-objective. Minimizing the number of transactions that have to communicate is shown to improve performance. Likewise, Quamar et al. (2013) optimize for the query span, i.e., the average number of machines involved in the execution of a query. By minimizing this meta-objective, which can easily be evaluated, the authors hope to indirectly reduce communication overhead, total resource consumption, and energy footprint. However, there is an obvious side-effect (discussed by the authors) of the query span objective that leads to higher response times for analytical queries that would benefit from a higher degree of parallelism.

The objective that we use to optimize performance is to minimize costly communication and to balance oversubscribed unbounded resources. The assumption is that network communication is a major cost factor that, given a partitioning, can

[4]Optimizing for communication volume is another valid meta-objective that is related but not identical to the message count.

be optimized by the allocation strategy[5]. The second assumption is that balancing load, especially for oversubscribed resources, leads to the best performance across all tasks. When the load is not balanced, single nodes receive more load than others and the tasks assigned to these nodes are bound to suffer from worse than possible performance.

3.3.2. Possible Constraints

The two basic formulations of the allocation problem focus on performance and resources. One formulation optimizes performance under the constraint of given resources. The other formulation optimizes required resources under performance constraints. On top of the basic formulations of the allocation problem, additional constraints can be incorporated, e.g., to guarantee availability or fault-tolerance (e.g., in Schaffner et al., 2013). Isolation, either from a security point of view or a performance point of view (e.g., Kiefer et al., 2014), may be another constraint. Aspects of maintainability and manageability may also be of importance and incorporated as constraints in the allocation problem.

Replication is often used to enforce some of the possible constraints, e.g., fault-tolerance. Curino et al. (2011), Kumar et al. (2014), and Schaffner et al. (2013) model replication in their variations of the allocation problem (details in Section 3.5). Replication of data has several aspects to consider. First, it needs to be decided which objects to replicate. Second, the number of replicas needs to be determined. Third, the execution platform needs to support replication (including read and write strategies). The first and the second decision are tradeoffs between the gain of additional replicas and the increased costs for update operations. If modeled in the workload graph, replication is almost independent of the allocation strategy with the only exception that it must be ensured that replicas of the same object are not mapped to the same node. We do not explicitly consider replication in this thesis because it is an orthogonal problem and because it is not a common feature found in all systems that we consider.

3.3.3. Problem Formulation

Given these preconditions, we use the following definition for an allocation and a corresponding formulation for the allocation problem that we try to solve in this thesis.

Definition 3.3 (Allocation)**.** An *allocation* is a mapping from a workload to an infrastructure. The mapping assigns vertices of the workload graph to vertices of the infrastructure graph. For each resource, a node's load is the combined weight (including non-linear behavior) of all workload vertices assigned to that node. An allocation is *valid* if the load is not greater than the capacity for all bounded resources and for all nodes. An allocation is *balanced* if the load is equal (within a tolerance) to the average load for all unbounded resources and for all nodes. An allocation's

[5] Note that the tradeoff between parallelism overhead and gain is part of the partitioning problem, not the allocation problem.

communication costs are the sum of all edge weights of edges between vertices that are assigned to different nodes.

Allocation Problem

Given a workload and an infrastructure, the allocation problem is to find a valid and balanced allocation that minimizes communication costs.

Allowing a node's load to be within a tolerance of the average load is necessary because it may not always be possible to find an allocation that perfectly balances the load. The acceptable degree of imbalance is a tuning parameter of the allocation strategy that has to be provided or empirically determined.

Our proposed solutions to the allocation problem are based on balanced k-way min-cut graph partitioning strategies and will be described in detail in Chapters 4 and 5.

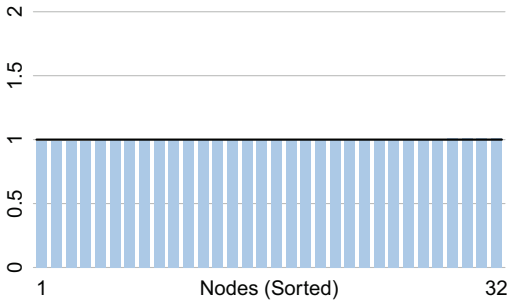
3.3.4. Relations to the Partitioning Problem

Given an allocation strategy, the question remains how to address the partitioning problem. With a workload, an infrastructure, and an allocation, we sketch how to incrementally update the partitioning. Incremental updates to the partitioning are based on two basic operations: (1) splitting of partitions and (2) merging of partitions (both operations can easily be applied to the workload graph). The costs assigned to vertices in the workload graph help to determine candidates for the split operation. Partitions that consume high amounts of processing or memory resources may benefit from a higher degree of parallelism. On the other hand, partitions from the same relation that are assigned to the same node by the allocation strategy are candidates for the merge operation. Given the fact that the allocation strategy tries to minimize communication, the co-location of partitions of the same relation happens implicitly if the degree of parallelism is high. Other candidates for merges are workload vertices that are connected by edges with high edge costs.

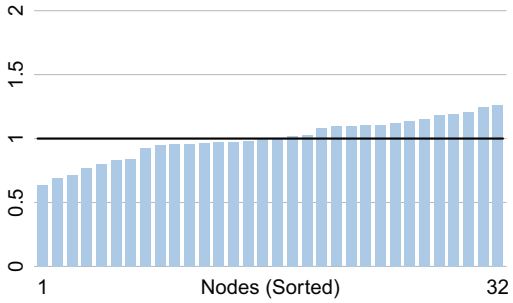
Given the candidates, whether or not it is beneficial to perform a split or merge operation is a challenging decision in the partitioning problem.

3.3.5. The Case for Individual Resources

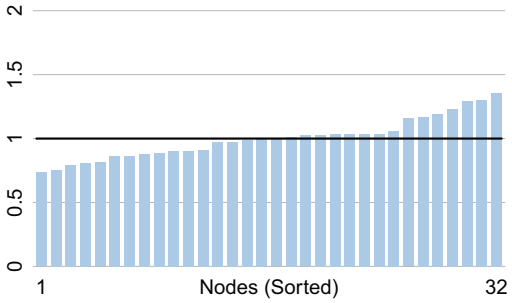
Our infrastructure model contains multiple resources and the allocation problem is formulated such that resources are treated individually, i.e., checked for capacity or balance. Other approaches to the allocation problem propose to combine all resources in a single resource that can then be optimized (e.g., Curino et al., 2011, detailed in Section 3.5). However, system resources like CPU or memory are only loosely coupled. Tasks may have a high processing cost with virtually no memory access costs, e.g., when the working set of the task fits in the cache. On the other hand, tasks that perform scans have high memory transfer costs but low processing costs. To illustrate the problem with combining resources, we conduct an experiment with two resources and show the results in Figure 3.7 (details about the



(a) Node Utilization According to the Sum of Weights



(b) Node Utilization (Weight 1)



(c) Node Utilization (Weight 2)

Figure 3.7.: Relative Node Utilizations for the Sum of Weights and Individual Weights (Node Utilization Normalized to the Average Node Utilization)

experiment are presented in Section 4.5.2). The workload in the experiment contains two resources, i.e., two weights, and the allocation strategy balances the sum of both resources. Figure 3.7a shows the resulting node utilization according to the sum of both resources. However, as shown in Figures 3.7b and 3.7c, the node utilizations are skewed with respect to the individual resources.

As a consequence, an allocation strategy must be able to handle multiple resources individually.

3.4. Requirements for Allocation Strategies

The allocation strategy tries to find a mapping that solves the allocation problem. Given the workload model and the infrastructure model, a number of requirements can be derived for allocation strategies.

Non-Linear Performance The infrastructure model assumes that performance does not scale linearly with the amount of work. The allocation strategy must be able to handle this non-linear infrastructure behavior.

Heterogeneous Infrastructure The allocation strategy must consider heterogeneous nodes in the infrastructure, i.e., nodes with different capacities. Furthermore, the infrastructure may be heterogeneous with respect to the communication net-

work, i.e., the network graph may not be fully connected and the links may have different capacities.

Multiple Individual Resources The allocation strategy must be able to handle multiple resources individually.

Bounded and Unbounded Resources The allocation strategy must be able to handle bounded resources, i.e., produce valid allocations, and unbounded resources.

Processing and Communication Costs The allocation strategy must consider processing costs, i.e., vertex weights, and communication costs, i.e., edge weights. Communication costs occur when tasks communicate to execute the workload.

Incremental Updates The allocation strategy must be able to incrementally update a solution after changes in the workload or the infrastructure occurred. The update strategy must consider and minimize migration costs induced by moving tasks between nodes. Naïvely computing a new allocation from scratch is prohibitively expensive with respect to migration costs. In presence of frequently changing workloads or infrastructures, incremental updates to the allocation should be cheap to compute.

3.5. Related Approaches to the Allocation Problem

In this section, we discuss closely related approaches to the allocation problem. 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. Consequently, the respective solution strategies use different algorithms. Given the different assumptions, a direct comparison with other approaches ranges from unfair to impossible. Instead, we try to identify the major differences in each approach that separate it from our work. Table 3.1 summarizes our findings.

Relational Cloud (Kairos) As part of the Relational Cloud Project[6], Curino et al. (2011) investigate workload-aware database monitoring and consolidation. The presented Kairos system consolidates databases based on the predicted combined resource utilization. The problem is termed differently by the authors (*consolidation problem*), but they seek a mapping from workloads to physical nodes that is comparable to the goal of the allocation problem.

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 models the workload as a time-series and seeks a mapping that is optimal for several time intervals, thereby avoiding costly migration. However, Kairos does

[6] `http://relationalcloud.com/`

	Kairos	Schism	SWORD	RTP	SharedDB
Non-Linear Performance	✓	✗	✗	✓	✗
Heterogeneous Infrastructures	✓	✗	✗	✓	✗
Individual Resources	✗	✗	✗	✗	(✓)
Communication Costs	✗	✓	✓	✗	✗
Incremental Updates	✗	✗	✓	✓	✗

Table 3.1.: Overview of Related Approaches to the Allocation Problem

not provide means to incrementally update a partitioning after the workload has changed.

Unlike the allocation problem in this thesis, 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 machines with the constraint of not overcommitting any single machine. To achieve this goal, the consolidation problem is modeled as a mixed-integer non-linear optimization problem and a general-purpose global optimization algorithm is used to solve the problem. The authors admit that this general approach is expensive and propose domain specific optimizations. With their improvements, the authors still report optimization times in the range of minutes for the test workloads.

The Kairos system has no notion of unbounded resources and overcommitting certain resources cannot easily be accomplished in the given problem formulation.

Relational Cloud (Schism) Schism (Curino et al., 2010) 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.

In Schism, the workload graph contains single tuples and it is decided on a per-tuple basis whether or not to replicate the tuple and which partition(s) to assign the replicates to. A major contribution of Schism focuses on the routing mechanism, i.e., Schism uses machine learning algorithms to compress the lookup table that maps tuples to partitions into more efficient range predicates.

The infrastructure model in Schism is simpler compared to this thesis, heterogeneous or non-linear hardware are 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. Following the assumptions made by the authors, Schism does not consider balancing more than one resource at a time.

With respect to the workload model, Schism assumes a static workload and does not provide any means to incrementally update a partitioning.

SWORD: Scalable Workload-Aware Data Placement and Replication Quamar et al. (2013) picked up the idea of Schism and present a project called SWORD. The authors propose a number of techniques to achieve higher scalability and to increase tolerance in presence failures and workload changes. For instance, the authors use hypergraph compression techniques to reduce the overhead of the graph partitioning step and propose incremental repartitioning to mitigate performance degradation due to workload changes.

In a follow-up to SWORD (Kumar et al., 2013, 2014), the authors introduce the notion of the *query span*, i.e., the average number of machines involved in the execution of a query. By minimizing the query span, the authors aim at reducing (1) communication overhead, (2) total resource consumption, (3) energy footprint, and (4) cost of distributed transactions. It should be noted that SWORD reduces the degree of parallelism and consequently may increase response times for analytical queries. The authors' argument is that response times are not critical in batch-executions of analytical workloads and that resource consumption is the metric that should be optimized.

The infrastructure model in SWORD does not consider heterogeneous or non-linear hardware.

Robust Tenant Placement and Migration (RTP) Schaffner et al. (2013) introduce the Robust Tenant Placement and Migration Problem (RTP). Unlike the allocation problem in thesis, the RTP tries to minimize the number of servers required by consolidating in-memory databases. The constraint is a guaranteed performance perceived by the user, measured in query response times. Additional constraints that are modeled into the RTP are that each database is replicated and that a single server may fail without violating performance guarantees. Unlike static approaches to data placement, 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.

The authors present multiple heuristics for the static RTP based on greedy algorithms and tabu search. The incremental RTP is solved by repeatedly solving the static RTP.

In the formalization of the RTP, tenants are modeled with size, i.e., required DRAM capacity, and load. The latter is a combined metric for CPU and memory bandwidth consumption. The authors argue that a single combined load metric can sufficiently model in-memory databases. As a consequence, the proposed solution algorithms for the RTP cannot easily be modified to handle individual resources. Distributed databases are intentionally not considered in the RTP and communication costs are omitted with the exception of migration costs. Heterogeneous servers are inherently supported in the problem formalization and the authors state that non-linear behavior can be respected given the availability of an estimator for combined resources consumption.

Deployment of Query Plans on Multicores (SharedDB) Giceva et al. (2014) investigate the deployment of query plans in a multicore machine. Although the results are presented for the shared-works system SharedDB (Giannikis et al., 2012)

which has a fundamentally different query execution model, they are interesting as they consider the allocation problem on a NUMA architecture. Some of the proposed modeling and solution approaches are interesting in a more general context. SharedDB uses a dataflow of shared, always-on relational operators. These operators are active throughout the whole execution of the workload and are shared among the concurrently executing queries. The goal of the plan deployment is to find a mapping of plan operators to cores (considering NUMA characteristics) that minimizes the amount of resources used while maintaining performance guarantees. The means of the plan deployment are temporal and spatial scheduling. Temporal scheduling tries to co-locate operators that (given the dataflow) cannot interfere with one another. Spatial scheduling tries to co-locate operators that use different resources (CPU or memory bandwidth) and operators that heavily communicate.

The authors introduce the notion of *resource activity vectors* to model CPU utilization and memory bandwidth utilization. The solution strategy proposed by the authors uses a two-step bin-packing algorithm where operators are first combined based the CPU utilization and in a second step, clusters of operators are combined based on memory bandwidth utilization.

The solution is interesting because it individually optimizes for the two modeled resources considering the characteristics of the underlying system. However, the bin-packing approach cannot easily be generalized to a higher number of resources or to the notion of unbounded resources. Furthermore, non-linear and heterogeneous hardware are additional characteristics of our infrastructure model that cannot easily be added to the proposed solution framework.

3.6. Summary

In this chapter, we first abstracted from actual data-oriented systems. A weighted graph was used to describe the infrastructure, which models individual bounded and unbounded resources with possibly non-linear behavior. Accordingly, we proposed to use a weighted graph to model the workload, which consisting of tasks that are connected by data transfers.

Based on the infrastructure and the workload model, we formulated the allocation problem and discussed different objectives and constraints. The allocation problem in this thesis seeks a valid and balanced allocation that minimizes communication costs.

Given the allocation problem, we derived several requirements for an ideal allocation strategy and evaluated related approaches with respect to these requirements.

The next chapter presents our proposed solution for the allocation problem. Our allocation strategy uses balanced k-way min-cut graph partitioning to partition the workload graph and assigns graph partitions to infrastructure nodes.

4. Balanced K-Way Min-Cut Graph Partitioning

In this chapter, we describe the balanced k-way min-cut graph partitioning problem (GPP)[1] and methods to solve it. We furthermore show that the GPP can be used to partition the previously introduced workload graph. The workload partitions can be mapped to the infrastructure graph to create a solution for the allocation problem. Throughout the chapter, we detail necessary modifications that enable graph partitioning methods to solve variations of the allocation problem.

The general goal of the balanced k-way min-cut problem is to partition a given graph into k parts such that the sum of edges that are cut is minimized while keeping the sizes of all parts within balance. Applied to the workload graph, a balanced min-cut partitioning minimizes communication and at the same time balances load across the nodes.

To develop a solution strategy for the allocation problem, we start with simplified formulations of infrastructure, workload, and allocation. Initially, in the GPP in Section 4.1, we assume a homogeneous infrastructure with only a single unbounded resource that behaves linearly. These limitations will be relaxed throughout this chapter and the following chapter. Section 4.2 adds multiple individual resources to the graph partitioning in the Multi-Constraint GPP (MC-GPP). Both problems, GPP and MC-GPP, are known and have been studied elsewhere and are here presented as foundations for our own modifications of the problem. Sections 4.3 and 4.4 present the Penalized GPP (P-GPP) and the Secondary Weight GPP (SW-GPP), which extend the graph partitioning algorithms to reflect non-linear resources. In Section 4.5, we experimentally evaluate benefit and applicability of an allocation strategy that is based on multi-constraint and penalized graph partitioning. We present results from synthetic partitioning experiments and a number of scalability experiments to support our approach.

Extensions to the core graph partitioning algorithms, which will be presented in Chapter 5, further relax limitations of the allocation strategy. Section 5.1 proposes methods to incrementally update an allocation. Section 5.2 deals with heterogeneous infrastructures and Section 5.3 proposes modifications to handle bounded resources.

4.1. Graph Partitioning

Partitioning a graph into k partitions of roughly equal size such that the total cut is minimized is NP-complete (Hyafil and Rivest, 1973). There exist approximation

[1] Note that the *graph partitioning problem* is different from the previous *data partitioning problem*. The data partitioning problem seeks a partitioning of a relation while the graph partitioning problem seeks subsets of the vertices of a graph.

algorithms with proven boundaries for variations of the problem. However, these results and the implementations of the methods are only of theoretical interest due to extremely long runtimes (Andreev and Racke, 2004; Buluç et al., 2013). Heuristics are used in practice to solve the GPP. Given the various applications, e.g., in scientific computation, road networks, or VLSI physical design, graph partitioning methods are well studied and have matured in the last two decades. A comprehensive overview of different graph partitioning methods can for instance be found in Buluç et al. (2013) or Bichot and Siarry (2011).

We formalize the graph partitioning problem in Section 4.1.1 before we describe the multilevel partitioning framework, a widely recognized solution heuristic, in Section 4.1.2. We concentrate on an instance of the multilevel partitioning framework that (1) provides good solutions very fast and (2) lends itself to being modified to solve the allocation problem (e.g., Buluç et al., 2013; Karypis and Kumar, 1995). Other variations of the framework, e.g., ones that combine multi-level partitioning with evolutionary algorithms, can provide better solutions at the price of increased complexity (Buluç et al., 2013; Sanders and Schulz, 2013). Global partitioning algorithms, i.e., algorithms that are not based on multilevel partitioning, such as the spectral partitioning algorithm, only work for small graphs and are hence not further discussed (Buluç et al., 2013).

4.1.1. Prerequisites

Problem 4.1 (GPP)**.** 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.

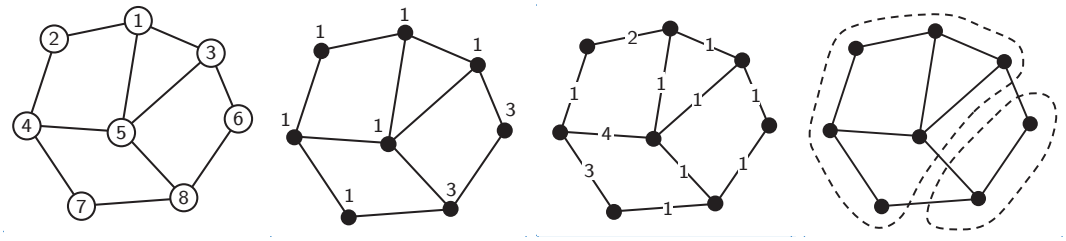
Minimizing the edge cut can be considered the objective in the GPP while the requirement that partitions are of similar size can be considered the constraint. The following definitions are used to formalize the problem and to describe its solution heuristics in detail. The definitions are accompanied by examples.

Definition 4.1 (Weighted Graph)**.** Let $G = (V, E, w_V, w_E)$ be an undirected, weighted graph with a set of vertices V, a set of edges E, and weight functions w_V and w_E. Vertex and edge weights are positive real numbers:

$$w_V \colon V \to \mathbb{R}_{>0} \text{ and}$$
$$w_E \colon E \to \mathbb{R}_{>0}.$$

The weight functions are naturally extended to sets of vertices and edges:

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



(a) Vertex Identifiers (b) Vertex Weights (c) Edge Weights (d) Partitioning

Figure 4.1.: Example of an Undirected Weighted Graph

v	$w_V(v)$	v	$w_V(v)$	e	$w_E(e)$	e	$w_E(e)$	e	$w_E(e)$
1	1	5	1	$\{1,2\}$	2	$\{3,5\}$	1	$\{5,8\}$	1
2	1	6	3	$\{1,3\}$	1	$\{3,6\}$	1	$\{6,8\}$	1
3	1	7	1	$\{1,5\}$	1	$\{4,5\}$	4	$\{7,8\}$	1
4	1	8	3	$\{2,4\}$	1	$\{4,7\}$	3		

Table 4.1.: Weight Functions w_V and w_E of Example Graph G_1

Graph Example. Let $G_1 = (V, E, w_V, w_E)$ be an exemplary graph. A visual representation of V_1 is shown in Figure 4.1. For G_1, the sets V and E are as follows:

$$V = \{1, 2, 3, 4, 5, 6, 7, 8\} \text{ and}$$

$$E = \{\{1,2\}, \{1,3\}, \{1,5\}, \{2,4\}, \{3,5\}, \{3,6\}, \{4,5\}, \{4,7\}, \{5,8\}, \{6,8\}, \{7,8\}\}.$$

The weight functions w_V and w_E are listed in Table 4.1.

Definition 4.2 (Graph Partitioning)**.** Let $\Pi = (V_1, \ldots, V_k)$ be a *partitioning* of V into k partitions $V_1, \ldots, V_k$ such that:

$$V_1 \cup \cdots \cup V_k = V \text{ and}$$
$$V_i \cap V_j = \emptyset \quad \forall i \neq j.$$

Definition 4.3 (Cut)**.** Given a partitioning, an edge that connects partitions is called a *cut edge*. The set E_{ij} is the set of cut edges between two partitions V_i and V_j:

$$E_{ij} := \{\{u, v\} \in E | u \in V_i, v \in V_j, i \neq j\}.$$

The set E_c is the set of all cut edges in a graph:

$$E_c := \bigcup_{i<j} E_{ij}.$$

The objective of the GPP is to minimize the *total cut* $w_E(E_c)$, i.e., the aggregated weight of all cut edges.

Definition 4.4 (Balance Constraint)**.** A *balance constraint* demands that all partitions have about equal weights. Let μ be the average partition weight:

$$\mu := \frac{w_V(V)}{k}.$$

For a graph partitioning to be balanced it must hold that

$$\forall i \in \{1, \ldots, k\} \colon w_V(V_i) \leq (1 + \epsilon) \cdot \mu,$$

where $\epsilon \in \mathbb{R}_{\geq 0}$ is a given imbalance parameter.

Perfectly balanced partitions (i.e., $\epsilon = 0$) are in many applications hard or even impossible to obtain. Therefore, ϵ is a parameter that can be used in the partitioning algorithm and hence in the allocation strategy to specify a tolerable degree of imbalance. The exact value of ϵ depends on the application and must be provided or empirically determined. Even with a larger imbalance parameter, there may not always exist a solution to the GPP. Especially small graphs can leave too few degrees of freedom to find a balanced partitioning.[2]

Graph Example (continued)**.** Figure 4.1d shows an example partitioning Π_1 of G_1 into two partitions that minimizes the cut and balances the partitions. The partitioning $\Pi_1 = (V_1, V_2)$ of G_1 is given by:

$$V_1 = \{1, 2, 3, 4, 5, 7\} \text{ and}$$
$$V_2 = \{6, 8\}.$$

Cut edges in G_1 under partitioning Π_1, i.e., edges that connect partitions, are $\{1, 3\}$, $\{5, 8\}$, and $\{7, 8\}$. Since there are only two partitions, the total edge cut $w_E(E_c)$ is 3 given that:

$$E_c = E_{12} = \{\{1, 3\}, \{5, 8\}, \{7, 8\}\}.$$

The average partition weight in G_1 is 6: $\mu = w_V(V)/k = 12/2 = 6$. Since, $w_V(V_1) = w_V(V_2) = \mu = 6$, the balance constraint is fulfilled for any $\epsilon \geq 0$.

Definition 4.5 (Neighborhood and Boundary Vertex)**.** The *neighbors* Γ of a vertex v are all the vertices connected to v via an edge:

$$\Gamma(v) := \{u | \{u, v\} \in E\}.$$

Given a partitioning, the *internal neighbors* Γ' of $v \in V_i$ are the neighbors of v that belong to the same partition:

$$\Gamma'(v) := \Gamma(v) \cap V_i.$$

Given a partitioning, the *external neighbors* Γ_j of $v \in V_i$ with respect to V_j are the neighbors of v that belong to a different partition V_j:

$$\Gamma_j(v) := \Gamma(v) \cap V_j.$$

A vertex $v \in V_i$ is called a *boundary vertex* if it has a neighbor in a different partition, i.e., $\exists_j \colon \Gamma_j(v) \neq \emptyset$.

[2] Our partitioning algorithms still yield partitionings in these cases and warn about the higher than expected imbalance. However, during all our experiments using $\epsilon = 0.03$ (i.e., at most 3% imbalance), we never encountered a graph that could not be balanced.

Graph Example (continued)**.** Let v be the vertex with identifier 5 in G_1, then the neighbors of v are $\Gamma(v) = \{1, 3, 4, 8\}$. Given Π_1, the internal neighbors of v are $\Gamma'(v) = \{1, 3, 4\}$ and the external neighbors with respect to V_2 are $\Gamma_2(v) = \{8\}$. The boundary vertices in G_1 under Π_1 are $\{3, 5, 6, 7, 8\}$.

Definition 4.6 (Vertex Degree and Gain)**.** 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:

$$id(v) := w_E(\{\{u, v\} \in E | u \in \Gamma'(v)\}).$$

Given a partitioning, 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:

$$ed_j(v) := w_E(\{\{u, v\} \in E | u \in \Gamma_j(v)\}.$$

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:

$$g_j(v) := ed_j(v) - id(v).$$

Note that the gain may be negative, which implies that moving the vertex increases the cut.

Graph Example (continued)**.** Given Π_1, the internal degree of v in G_1 is $id(v) = 6$ and the external degree of v with respect to V_2 is $ed_2(v) = 1$. The gain of v with respect to V_2 is $g_2(v) = -5$, hence moving v from V_1 to V_2 would increase the total cut by 5.

4.1.2. Multilevel Graph Partitioning Framework

Multilevel graph partitioning is rather a strategy or framework than a concrete method because it allows the use of different methods in its various steps. Multilevel graph partitioning 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. Different strategies and optimizations exist for all three phases. Figure 4.2 shows an overview of the steps.

The idea of multilevel graph partitioning is to reduce the initial problem to a much smaller problem with similar characteristics, then to solve the small problem using a (potentially expensive) method, and last to project the solution of the small problem to the initial problem. To improve the quality of the final solution, each uncoarsening step is followed by a refinement step. The motivation of this refinement step is to improve the partitioning, e.g., by local vertex swapping heuristics, that was projected from the coarse graph before it is projected to the next finer level.

There are several intuitive reasons why the multilevel approach leads to very good results. First, one can use very expensive algorithms or a variety of algorithms on the coarse level without increasing the overall execution time by a lot. Second, coarsening broadens the scope of local optimization algorithms, i.e., moving vertices between partitions in the coarse graph leads to potentially long distance moves of

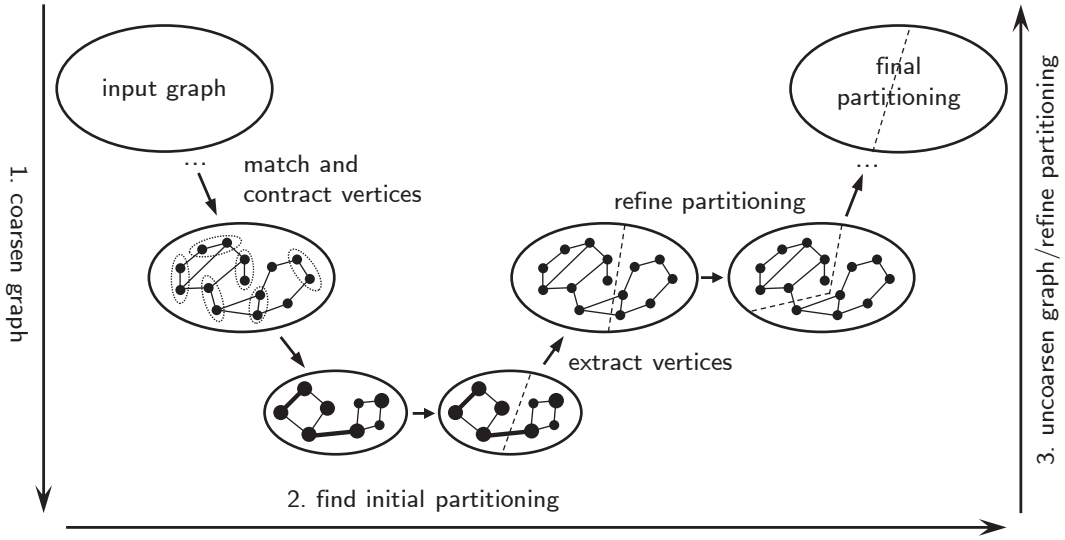


Figure 4.2.: Multilevel Graph Partitioning Framework

many vertices in the finest graph. Third, local improvements during the uncoarsening are fast because they start with an already good solutions.

Two general strategies exist for computing a k-way partitioning using the multilevel paradigm, (1) direct k-way partitioning and (2) recursive bisection (Karypis and Kumar, 1998a). The first strategy directly partitions the graph into the desired number of partitions, while the second approach repeatedly partitions a (sub)graph into two partitions. A major difference between the two strategies is that for direct k-way partitioning coarsening and uncoarsening are only performed once while these steps have to be performed multiple times (once for each bisection) in the second strategy. A second difference is that refinement operations have global knowledge when used on a k-way partitioning while they can only see a subgraph in all but the topmost bisection steps. Experiments showed that multilevel k-way partitioning produces partitionings with a quality that is comparable to partitionings produced by multilevel recursive bisection while requiring substantially less time (Karypis and Kumar, 1998a). The greatest drawback of direct k-way partitioning is that refinement algorithms are usually more complicated compared to the corresponding algorithms for bisection. The initial partitioning of the coarsest graph (step 2 in the framework) is independent of the global multilevel partitioning strategy. A heuristic can use direct k-way partitioning globally but less complicated recursive bisection algorithms to find an initial partitioning.

The following paragraphs describe the three phases of the multilevel graph partitioning framework in more detail.

Graph Coarsening

In the coarsening phase, a series of smaller graphs is derived from the input graph by collapsing adjacent pairs of vertices such that cuts in the coarse graph reflect cuts in the fine graph. The goal is to gradually approximate the original problem

and the input graph with fewer degrees of freedom.

Coarsening is commonly implemented by contracting a subset of vertices $U \subset V$ and replacing it with a single vertex u' having the weight

$$w_V(u') := \sum_{u \in U} w_V(u).$$

Contraction might produce parallel edges which are replaced by a single edge with the accumulated weight of the parallel edges, i.e., for any given v in $V \setminus U$:

$$w_E(\{u', v\}) := \sum_{u \in U, \{u,v\} \in E} w_E(\{u, v\}).$$

Contracting vertices like this implies that balanced partitions on the coarse level represent balanced partitions on the fine level with the same cut value.

Different strategies exist to select matchings of vertices to be contracted during a coarsening step. The minimalistic approach is to contract a single pair in every step, leading to almost $|V|$ levels. Another strategy is to contract maximal matchings, given that each vertex can be used at most once per matching. This leads to a logarithmic number of levels. 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). Different heuristics are used in practice to select vertices to be contracted, e.g., heavy edge matching, random matching, or path growing algorithm (Buluç et al., 2013; Drake and Hougardy, 2003).

The coarsening ends when the coarsest graph has at most a few hundred vertices and is hence sufficiently small to be initially partitioned.

Initial Graph Partitioning

Many different algorithms exist to find an initial partitioning (Buluç et al., 2013). Like the global multilevel partitioning strategy, methods for the initial partitioning are either based on direct k-way partitioning or on recursive bisection. For some algorithms there exist both implementations, a direct k-way partitioning and a bisection-based partitioning.

A simple but effective method to find an initial partitioning is greedy graph-growing. For bisection, the algorithm starts with a random vertex. This 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. In this thesis, we focus on graph-growing for the initial partitioning as it is simple yet effective and it can be modified to the needs at hand.

A number of evolutionary methods and metaheuristics have been proposed that yield better initial partitioning at the price of long runtimes (e.g., Buluç et al., 2013).

The method that finds an initial partitioning can be used together with any number of refinement strategies that try to improve that partitioning. Different refinement strategies are introduced in the next paragraph where they are used during the uncoarsening phase.

Graph Uncoarsening and Refinement

When the initial partitioning is found, the solution can repeatedly be mapped to the next finer graph and ultimately to the initial problem. At each level, the coarse graph is transformed to a finer graph by assigning previously contracted vertices to the same partition. The extraction of vertices is followed by a refinement step. Uncoarsening and refinement are repeated until the finest graph, i.e., the initial graph, is reached.

A variety of refinement methods are used in practice (Buluç et al., 2013). Methods range from fine-grained local vertex swapping heuristics to methods that try to globally improve the partitioning. Refinement methods can be used to improve the total cut or the balance of the partitions. Ideally, one metric is improved while the other one does at least not degrade. However, to leave local minima it may be beneficial to accept temporary decline in one of the metrics. Interleaving and repeatedly executing refinement methods that try to improve balance and such that try to improve the total cut can also increase the quality of the final partitioning.

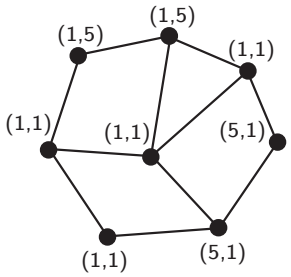
Local vertex swapping is a refinement metaheuristic that can be parameterized with different strategies to select vertices to move. Kernighan and Lin (1970) proposed to move the vertex with the highest value of the gain function, i.e., the vertex that yields the largest decrease in total cut. Each vertex 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. Fiduccia and Mattheyses (1982) improved the KL method with carefully designed data structures. Also, they allow sequences of moves as long as the entire sequence improves the total cut. This method, known as KL/FM, is widely used in practice as it is simple and hence fast while leading to very good results, especially when starting from an already good partitioning. Karypis and Kumar (1998b) further accelerated KL/FM by only allowing boundary vertices to move and by stopping a round when the total cut does not improve after a number of moves. They also introduce random tie breaking and additional rounds even when no improvements have been found to further improve the result. Sanders and Schulz (2013) allow local moves to even degrade the balance as long as a combination of moves globally improves balance. Helpful sets, introduced by Diekmann et al. (1995) extend the vertex swapping to sets of vertices.

All vertex-swapping techniques are originally proposed to improve a bisection. However, all have been extended to directly improve a k-way partitioning as well.

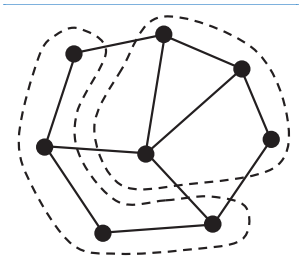
Other improvement strategies like tabu search, flow-based improvement, random-walk, or methods based on diffusion are beyond the scope of this thesis since they are often too complex to be modified as needed (see Buluç et al. (2013) for further references).

4.2. Multi-Constraint Graph Partitioning

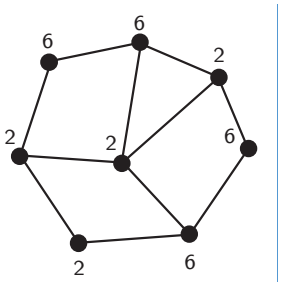
The GPP seeks a partitioning of the graph that minimizes the total cut and balances the partition weights. However, a single weight per vertex is insufficient to partition workloads that model various resources. Solving the GPP with the intention of balancing multiple resources, i.e., vertex weights, individually has been studied as



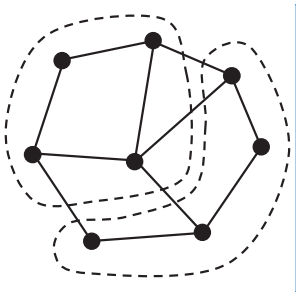
(a) Multiple Vertex Weights



(b) Multi-Constraint Partitioning



(c) Combined Vertex Weights



(d) Single-Constraint Partitioning

Figure 4.3.: Example of a Graph Partitioning with Multiple and Combined Vertex Weights

Multi-Constraint Graph Partitioning Problem, e.g., by Karypis and Kumar (1998a). The necessary definitions and a solution strategy for the multi-constraint graph partitioning problem are presented in the subsequent sections.

4.2.1. Prerequisites

Problem 4.2 (MC-GPP)**.** The Multi-Constraint Graph Partitioning Problem (MC-GPP) is a generalization of the GPP where each vertex has m weights assigned and the goal is to compute a k-way partitioning with minimal cut such that each one of the m weights is individually balanced within a specified tolerance.

A tempting approach, using the existing solutions for the GPP, is to sum up all resources (i.e., vertex weights) to get a single vertex weight that can be balanced. A linear combination of weights can be used instead of the sum to prioritize certain resources over others. However, as will be shown in an example shortly, balancing the combined weight can still lead to significant skew in the individual components which violates the requirement of individually balanced weights in the MC-GPP.

Multi-Constraint Graph Example. Figure 4.3a shows a graph with two weights per vertex (vertex identifiers and edge weights remain the same as in the previous Graph Example shown in Figure 4.1). Solving the MC-GPP leads to the partitioning with the total cut of 8 shown in Figure 4.3b. Both partitions have a weight of $(8, 8)$ and are hence perfectly balanced. Figure 4.3c shows the same graph where

the individual vertex weights are summed up to get a single weight per vertex. Figure 4.3d shows that partitioning this graph leads to different partitions. Although this partitioning has a smaller total cut of 6, when applied to the original graph, it leads to partition weights $(4, 12)$ and $(12, 4)$ respectively. Hence, the partitions are individually skewed by a factor of 3.

To formalize the MC-GPP, two of the previous definitions need to be modified to reflect the arbitrary number of balancing constraints.

Definition 4.7 (MC Weighted Graph)**.** Let $G = (V, E, (w_V^1, \ldots, w_V^m), w_E)$ be an undirected, weighted graph as in Definition 4.1. The single vertex weight function is replaced by a vector of size m of vertex weight functions w_V^i. Vertex weights are positive real numbers:

$$w_V^i \colon V \to \mathbb{R}_{>0} \text{ for } i = 1, \ldots, m.$$

The weight of a single vertex is a vector of size m:

$$w_V(v) := (w_V^1(v), \ldots, w_V^m(v)).$$

The weight functions are naturally extended to sets $V' \subseteq V$ of vertices:

$$w_V^i(V') := \sum_{v \in V'} w_V^i(v) \text{ for } i = 1, \ldots, m,$$
$$w_V(V') := (w_V^i(V'), \ldots, w_V^i(V')).$$

Definition 4.8 (MC Balance Constraint)**.** A *balance constraint* for multiple vertex weights demands that for all weights individually all partitions have about equal weights. For $i = 1, \ldots, m$, let μ^i be the average partition weight of the i-th constraint:

$$\mu^i := \frac{w_V^i(V)}{k}.$$

For a graph partitioning to be balanced it must hold that

$$\forall i \in \{1, \ldots, m\} \forall j \in \{1, \ldots, k\} \colon w_V^i(V_j) \leq (1 + \epsilon^i) \cdot \mu^i,$$

where $\epsilon^i \in \mathbb{R}_{\geq 0}$ is a given imbalance parameter for the i-th constraint. A balanced partitioning with m weights is sometimes referred to as being *m-balanced*.

Multi-Constraint Graph Example (continued)**.** For the graph and the corresponding partitioning shown in Figures 4.3a and 4.3b respectively, the average partition weights are

$$\mu_1 = \frac{w_V^1(V)}{k} = \frac{1}{k} \sum_{v \in V} w_V^1(v) = 16/2 = 8 \text{ and}$$
$$\mu_2 = \frac{w_V^2(V)}{k} = \frac{1}{k} \sum_{v \in V} w_V^2(v) = 16/2 = 8.$$

Furthermore,

$$
\begin{aligned}
w_V^1(V_1) &= \sum_{v \in V_1} w_V^1(v) = 8, \\
w_V^2(V_1) &= \sum_{v \in V_1} w_V^2(v) = 8, \\
w_V^1(V_2) &= \sum_{v \in V_2} w_V^1(v) = 8, \text{ and} \\
w_V^2(V_2) &= \sum_{v \in V_2} w_V^2(v) = 8.
\end{aligned}
$$

Hence, the balance constraint is fulfilled for arbitrary imbalance parameters $\epsilon^1 \geq 0$ and $\epsilon^2 \geq 0$.

4.2.2. Multi-Constraint Graph Partitioning Algorithm

The multilevel graph partitioning framework can be used to find good solutions of the MC-GPP (Karypis and Kumar, 1998a). However, the building blocks of the framework, i.e., algorithms for coarsening, initial partitioning, and uncoarsening and refinement, need to be modified to reflect the multiple balancing constraints.

Graph Coarsening

To coarsen the graph, a matching of vertices is built and vertices are contracted according to the matching. The *heavy-edge* heuristic which successfully reduces exposed edge weight in the GPP can also be used in the MC-GPP to minimize the total cut of the initial partitioning and therefore the final solution. However, balancing multiple weights is hard to accomplish even in the small coarsest graph used for the initial partitioning. Therefore, the coarsening can also be used to balance the different weights of the vertices. The intuition is that if every vertex v has equal weights $w_V^1(v) = \cdots = w_V^m(v)$, then the m-weight balancing problem becomes balancing a single weight, a known problem from the GPP. Therefore the *balanced-vertex* heuristic tries to contract pairs of vertices that minimize the difference among the weights of the contracted vertex.

Studies show (Karypis and Kumar, 1998a) that using the heavy-edge heuristic leads to good total cuts but often fails to meet the balance constraints. Using the balanced-vertex heuristic on the other hand leads to balanced partitions but potentially bad total cuts. A combination of both can be used to find a solution with a good total cut that meets the balance constraint. For example, the heavy-edge heuristic is used to find candidates for collapsing and the balanced-vertex heuristic is used as tie breaker for edges of the same weight.

Initial Graph Partitioning

For the initial partitioning in the MC-GPP, recursive bisection is used as it is considerably easier to balance multiple weights in just two partitions than to balance

multiple weights in multiple partitions. Karypis and Kumar (1998a) propose a modified version of the greedy region-growing algorithm followed by additional balancing and refinement steps to find an m-balanced bisection.

The algorithm starts by assigning all but a random start vertex to partition V_1 and the random start vertex to V_2. Then, m priority queues are used (ordered by the decreasing value of the gain function) and each vertex is added to just one priority queue based the vertex' largest weight. Specifically, a vertex $v \in V_1$ is added to the i-th queue, if and only if $w_V^i(v) = \max(w_V(v))$. To grow partition V_2, the priority queue to pick a vertex from is selected depending on the relative weights of partition V_1, i.e., the algorithm tries to reduce the highest weight by moving a vertex to the other partition. Precisely, it picks the i-th priority queue if $w_V^i(V_1) = \max(w_V(V_1))$. Should this queue be empty, the queue according to the second highest weight in V_1 is picked. The algorithm stops when more than half of a single weight is assigned to partition V_2.

Graph Uncoarsening and Refinement

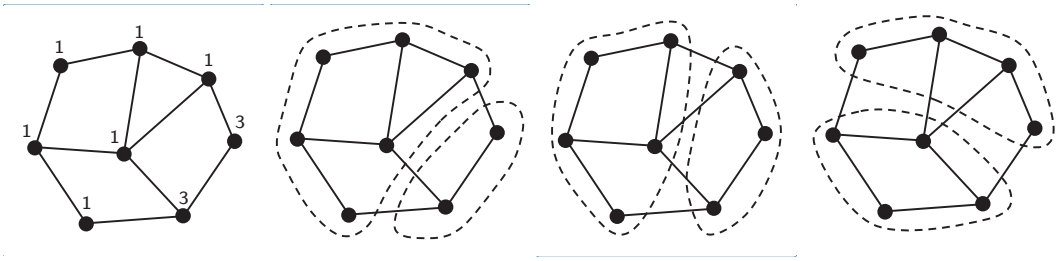
The KL/FM method can be modified to balance or refine multi-constraint graph partitionings. Here, we describe the KL/FM method for refining a bisection, the generalization to k-way refinement uses the same principles.

The m-weight KL/FM method uses $2m$ priority queues, ordered by the decreasing value of the gain function. Similar to the algorithm for the initial partitioning, a vertex v belongs to the i-th queue if the i-th weight is the highest weight of the vertex, i.e., $w_V^i(v) = \max(w_V(v))$. To move a vertex from one partition to the other, a queue is picked depending on the highest partition weight of both partitions. Should this queue be empty, then the queue according to the second highest weight of the same partition is used.

As in the original KL/FM method, a series of vertex moves is performed. When no more vertices can be moved, the point in the series of moves that has the lowest total cut (for refinement) or that has the best partition balance (for balancing) is selected and all moves up to this point are materialized.

4.3. Penalized Graph Partitioning

Solving the GPP or the MC-GPP yields an allocation in data-oriented systems. Partitioning the workload graph with minimal cut reduces costly communication and balancing the partitions leads to evenly utilized nodes. However, there is a mismatch between the graph partitioning problem 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 we have shown in the infrastructure model in Section 3.1, performance on actual hardware resources usually does not scale linearly with the number of concurrent tasks. To account



(a) Vertex Weights (b) No Penalty (c) Linear Penalty (d) Square Penalty

Figure 4.4.: Example of Graph Partitionings with Different Penalty Functions

for the non-linear behavior, we propose the *Penalized Graph Partitioning Problem* (P-GPP). The idea is to introduce a *penalized partition weight* and to modify the graph partitioning problem accordingly to balance the penalized partition weights.

The P-GPP assumes the special case of non-linear performance, i.e., a penalty function on top of the performance yielded by a linear model. Arbitrary non-linear performance models will be discussed in the Secondary Weight Graph Partitioning Problem in Section 4.4.

4.3.1. Prerequisites

Problem 4.3 (P-GPP)**.** The P-GPP is a generalization of the GPP or the MC-GPP where each vertex has one or m weights assigned and the goal is to find a k-way partitioning with minimal cut such that each one of the m penalized partition weights is individually balanced within a given tolerance.

Penalized Graph Example. Figure 4.4a shows again the graph from the first example (vertex identifiers and edge weights remain the same as in the Graph Example shown in Figure 4.1). Solving the GPP leads to the partitioning with the total cut of 3 shown in Figure 4.4b. 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 4.4c. 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 4.4d is the solution to the P-GPP. The partitioning obviously depends on the given penalty function.

To formulate the P-GPP, previous definitions need to be modified to reflect the penalized partition weights. For the sake of readability, we assume a single weight per vertex (i.e., $m = 1$) throughout this section to be able to omit the superscript in every formula. However, the definitions and operations translate naturally to multiple vertex weights.

Definition 4.9 (P Weighted Graph)**.** Let $G = (V, E, w_V, w_E, p)$ be an undirected, weighted graph as in Definition 4.1. Furthermore, let p be a positive, monotonic increasing penalty function that penalizes a partition weight based on the partition cardinality[3]:

$$p\colon \mathbb{N} \to \mathbb{R}_{\geq 0} \text{ with}$$

[3]In the case of multiple vertex weights, a vector of penalty functions $(p^1, \ldots, p^m)$ is used instead

$$p(n_1) \leq p(n_2) \text{ for } n_1 \leq n_2.$$

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

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

Penalized Graph Example (continued)**.** The example partitioning in Figure 4.4c uses a linear penalty function, i.e.,

$$p(|V|) := |V|.$$

Accordingly, using Definition 4.9, 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 4.4d uses a square penalty function, i.e.,

$$p(|V|) := |V|^2.$$

Accordingly, using Definition 4.9, the partition weights are

$$w_V(V_1) = \sum_{v \in V_1} w_V(v) + p(|V_1|) = 6 + 16 = 22 \text{ and}$$
$$w_V(V_2) = \sum_{v \in V_2} w_V(v) + p(|V_2|) = 6 + 16 = 22.$$

The simple example shows that the square penalty function, which grows quickly with the number of vertices in a partition, leads to evenly distributed vertices. The penalty in this example outweighs the actual partition weights.

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. Specifically, using definition 4.9 for two partitions V_1 and V_2:

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

For arbitrary penalty functions it holds that

of p. The individual weights can be penalized independently with possibly different penalty functions.

$$p(|V_1 \cup V_2|) \neq p(|V_1|) + p(|V_2|).$$

It follows that in general

$$w_V(V_1 \cup V_2) \neq w_V(V_1) + w_V(V_2). \tag{4.1}$$

Under condition 4.1, the total weight of all vertices is in general not equal to the total weight of all partitions. We therefore introduce the following definition of the two weights.

Definition 4.10 (Total Vertex Weight, Total Partition Weight)**.** 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.,

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

Penalized Graph Example (continued)**.** The example partitioning in Figure 4.4d illustrates a case where the total vertex weight is not equal to the total partition weight. Using the definition,

$$w_V = \sum_{v \in V} w_V(v) + p(|V|) = 12 + 64 = 76 \text{ and}$$
$$w_\Pi = \sum_{i=1}^{k} w_V(V_i) = 22 + 22 = 44.$$

It follows from the definition that the total partition weight of the graph is not constant anymore 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 definition for a balance constraint needs to be modified.

Definition 4.11 (P Balance Constraint)**.** A *balance constraint* for penalized partitions demands that all penalized partitions have about equal weights. Let μ be the average partition weight:

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

For a graph partitioning to be balanced it must hold that

$$\forall i \in \{1, \ldots, k\} \colon w_V(V_i) \leq (1 + \epsilon) \cdot \mu,$$

where $\epsilon \in \mathbb{R}_{\geq 0}$ is a given imbalance parameter.

4.3.2. Penalized Graph Partitioning Algorithm

We propose modifications of the multilevel graph partitioning algorithm to solve the P-GPP. First, we describe how basic operations that are used throughout the algorithm need to be modified to reflect partition penalties. Then, we will detail the necessary modifications to the three building blocks of the multilevel graph partitioning framework, namely coarsening, initial partitioning, and uncoarsening and refinement.

During graph partitioning and refinement, it is often necessary to move a vertex v from one partition (V_1) to another partition (V_2). For the sake of computational efficiency, the partition weights of the resulting partitions should be computed incrementally instead of from scratch.

Operation 4.1 (Moving a Vertex)**.** 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:

$$\begin{aligned} w_V(V_1') = w_V(V_1 \setminus v) &= \sum_{u \in V_1 \setminus v} w_V(u) + p(|V_1 \setminus v|) \\ &= \sum_{u \in V_1} w_V(u) - w_V(v) + p(|V_1| - 1) \\ &= w_V(V_1) - w_V(v) - p(|V_1|) + p(|V_1| - 1). \end{aligned}$$

Using the same intermediate steps, it follows for V_2':

$$w_V(V_2') = w_V(V_2 \cup v) = w_V(V_2) + w_V(v) - p(|V_2|) + p(|V_2| + 1).$$

Operation 4.2 (Combining Partitions)**.** 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:

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

Graph Coarsening

To coarsen the graph, a matching of vertices has to be determined and vertices have to be contracted accordingly. The previously introduced heavy-edge heuristic and the balanced-vertex heuristic can both 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 4.2. From the implementation point of view this implies that the number of vertices comprising a contracted vertex needs to be maintained. The number of vertices being contracted is referred to as the *cardinality* of a contracted vertex.

Initial Graph Partitioning

We propose 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 from one partition to the other partition has to use Operation 4.1 to calculate the partition weights after the vertex has been moved. Moreover, the stop condition of the region growing algorithm has to be modified to account for the new balance constraint (Definition 4.11). In the previous 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 4.1).

Graph Uncoarsening and Refinement

The penalties have to be considered during the uncoarsening and refinement of the graph. Similar to the modifications of the region growing algorithm, the KL/FM method has to use Operation 4.1 whenever a vertex is moved between partitions. Furthermore, when the KL/FM method is used to balance a partitioning, the modified balance constraint in Definition 4.11 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.

4.4. Secondary Weight Graph Partitioning

In our infrastructure model in Section 3.1, we introduced two types of non-linear behavior. The penalized resource usage assumes that the basic resource usage can be combined linearly and that contention causes a non-linear penalty above a certain load level. The penalized resource usage is covered in the P-GPP. The second type of non-linear behavior in the infrastructure model is general non-linear resource usage. Here, the assumption is that the non-linear performance can be modeled as an arbitrary function of underlying (linear) resources. To model this general non-linear behavior, we introduce the notion of *secondary weights* and propose the *Secondary Weight Graph Partitioning Problem* (SW-GPP). Secondary vertex weights are derived from primary vertex weights. Arbitrary non-linear functions (positive and monotonic increasing) can be used to derive secondary weights from one or multiple primary weights. The restriction to positive and monotonic increasing

functions is reasonable given the application. Adding more work to a partition should in general increase to induced load.

4.4.1. Prerequisites

Problem 4.4 (SW-GPP)**.** The SW-GPP is a generalization of the GPP or the MC-GPP where each vertex has one or multiple primary weights assigned and the goal is to find a k-way partitioning with minimal cut such that each of the derived secondary partition weights is individually balanced within a given tolerance.

To solve the SW-GPP, secondary weights need to be defined. For the sake of readability, we assume only a single secondary weight in the graph. However, the definitions extend naturally to multiple secondary weights.[4]

Definition 4.12 (Secondary Weight)**.** Let $G = (V, E, w_V, w_E)$ be an undirected, weighted graph as in Definition 4.1. Furthermore, let f be a positive, monotonic increasing function:

$$f\colon \mathbb{R} \to \mathbb{R}_{\geq 0} \text{ with}$$
$$f(x_1) \leq f(x_2) \text{ for } x_1 \leq x_2.$$

The *secondary weight* $w_{f,V}$ of a vertex is defined as:

$$w_{f,V}(v) := (f \circ w_V)(v) = f(w_V(v)).$$

The secondary vertex weight function is extended to sets $V' \subseteq V$:

$$w_{f,V}(V') := f(w_V(V')).$$

In the case of multiple primary vertex weights, secondary weights can also be derived from more than one primary weight.

Definition 4.13 (Secondary Weight, continued)**.** Let $G = (V, E, (w_V^1, \ldots, w_V^m), w_E)$ be a weighted graph with multiple weights per vertex as in Definition 4.7. Furthermore, let f be a function with N real arguments:

$$f\colon \mathbb{R}^N \to \mathbb{R}_{\geq 0}.$$

For the sake of readability, let N be equal to 2, i.e., the secondary weight is based on two primary weights. Given two primary weights w_V^i and w_V^j, the secondary weight $w_{f,V}$ of a vertex is then defined as:

$$w_{f,V}(v) := f(w_V^i(v), w_V^j(v)).$$

The secondary vertex weight function is extended to sets $V' \subseteq V$:

$$w_{f,V}(V') := f(w_V^i(V'), w_V^j(V')).$$

The function f must be such that the secondary weight $w_{f,V}$ is monotonic increasing:

$$w_{f,V}(V_1) \leq w_{f,V}(V_2) \text{ for } V_1 \subseteq V_2.$$

[4] Multiple secondary weights are comparable to multiple primary weights and the necessary changes in the definition are comparable to the changes in the definitions from using a single primary weight to using multiple primary weights in the GPP and the MC-GPP.

4.4.2. Secondary Weight Graph Partitioning Algorithm

Given the prerequisites, it becomes clear that the P-GPP is subsumed by the SW-GPP. Assume that w_V^1 is the original vertex weight and that w_V^2 is a trivial vertex weight that reflects the vertex' cardinality, i.e., $w_V^2(v) = 1$ for all $v \in V$. Defining a secondary weight $w_{f,V}$ based on the function

$$f(w_V^1(v), w_V^2(v)) := w_V^1(v) + p(w_V^2(v))$$

resembles the penalized graph partitioning problem. However, having the P-GPP as a special case is useful from a computational point of view. The P-GPP has properties that were used in Operations 4.1 and 4.2 to efficiently compute the weights of partitions that result from frequent operations, i.e., moving a vertex and combining partitions. Given the arbitrary nature of secondary weights, these optimizations cannot be used for the SW-GPP. The generalization of Operations 4.1 and 4.2 are as follows.

Operation 4.3 (Moving a Vertex with Secondary Weights)**.** When a vertex v is moved from partition V_1 to partition V_2, the secondary partition weights of the resulting partitions $V_1' := V_1 \setminus v$ and $V_2' := V_2 \cup v$ are as follows:

$$\begin{aligned} w_{f,V}(V_1') = w_{f,V}(V_1 \setminus v) = f(w_V(V_1 \setminus v)) &= f(\sum_{u \in V_1 \setminus v} w_V(u)) \\ &= f(\sum_{u \in V_1} w_V(u) - w_V(v)) \\ &= f(w_V(V_1) - w_V(v)). \end{aligned}$$

Using the same intermediate steps, it follows for V_2':

$$w_{f,V}(V_2') = w_{f,V}(V_2 \cup v) = f(w_V(V_2) + w_V(v)).$$

Operation 4.4 (Combining Partitions with Secondary Weights)**.** When two partitions V_1 and V_2 are combined, the secondary partition weight of the resulting partition $V' := V_1 \cup V_2$ can be calculated as follows:

$$\begin{aligned} w_{f,V}(V') = w_{f,V}(V_1 \cup V_2) &= f(\sum_{v \in V_1 \cup V_2} w_V(v)) \\ &= f(\sum_{v \in V_1} w_V(v) + \sum_{v \in V_2} w_V(v)) \\ &= f(w_V(V_1) + w_V(V_2)). \end{aligned}$$

The operations can be optimized by maintaining the primary as well as the secondary weight for each partition. However, the function f still needs to be evaluated every time a partition changes.[5]

[5] In contrast, the penalty function in the P-GPP has natural numbers between zero and $|V|$ as argument. Thus, all possible values of the penalty function can be pre-computed and stored to speed up frequent operations.

The solution to the SW-GPP is similar to the one for the P-GPP. The difference is that Operations 4.1 and 4.2 are replaced with Operations 4.3 and 4.4 in all steps of the multilevel graph partitioning framework.

4.5. Experimental Evaluation

In this section, we evaluate graph partitioning as an allocation strategy. We sketch our implementation of the penalized graph partitioning, which is based on the METIS graph partitioning tools. Furthermore, we evaluate the benefit of multi-constraint and penalized graph partitioning over standard graph partitioning in two experiments that use a synthetic workload and infrastructure. Last in the section, we confirm the applicability of graph partitioning as a solution for the allocation problem by presenting results of various scalability experiments.

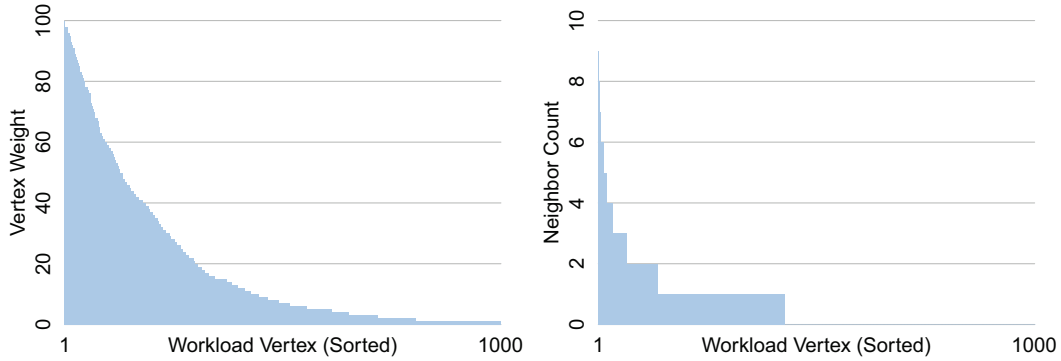
4.5.1. Penalized Graph Partitioning in METIS (PenMETIS)

METIS is a well known set of serial programs that, among other things, can be used for graph partitioning. METIS is developed by the research group of George Karypis at the University of Minnesota.[6] The algorithms implemented in METIS are based on multilevel recursive bisection, multilevel k-way partitioning, and multi-constraint partitioning.

For this thesis, we modified METIS (version 5.1) to support penalized graph partitioning (we termed the resulting tool PenMETIS). We decided to focus on penalized partitioning over secondary weight partitioning because it fits the infrastructure models in our example systems and because of the optimizations, e.g., incrementally updating partition weights, that are only possible with penalized graph partitioning. A number of modifications were necessary to support penalized partition weights in METIS. For instance, partition weights are assumed to be natural numbers in METIS. Using penalized partition weights, which can be arbitrary real numbers, required changing vertex and partition weights to real data types throughout the program. Furthermore, we added methods to provide and maintain a penalty function and modified the routines that calculate a partition's weight. We also added code to maintain vertex and partition cardinalities throughout the multilevel partitioning algorithm. The latter was necessary to implement new methods to move a vertex and to combine partitions, both using the total partition weight instead of the total vertex weight.

For PenMETIS, we focus on the serial version of METIS. Our scalability experiments show that the serial version is fast enough to partition even large workload graph. However, there is no reason why the same modifications that went into PenMETIS may not be incorporated in the parallel versions of METIS in the future.

[6] `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`



(a) Vertex Weight Distribution (b) Neighbor Count Distribution

Figure 4.5.: Workload Characteristics of the Synthetic Workload Graph

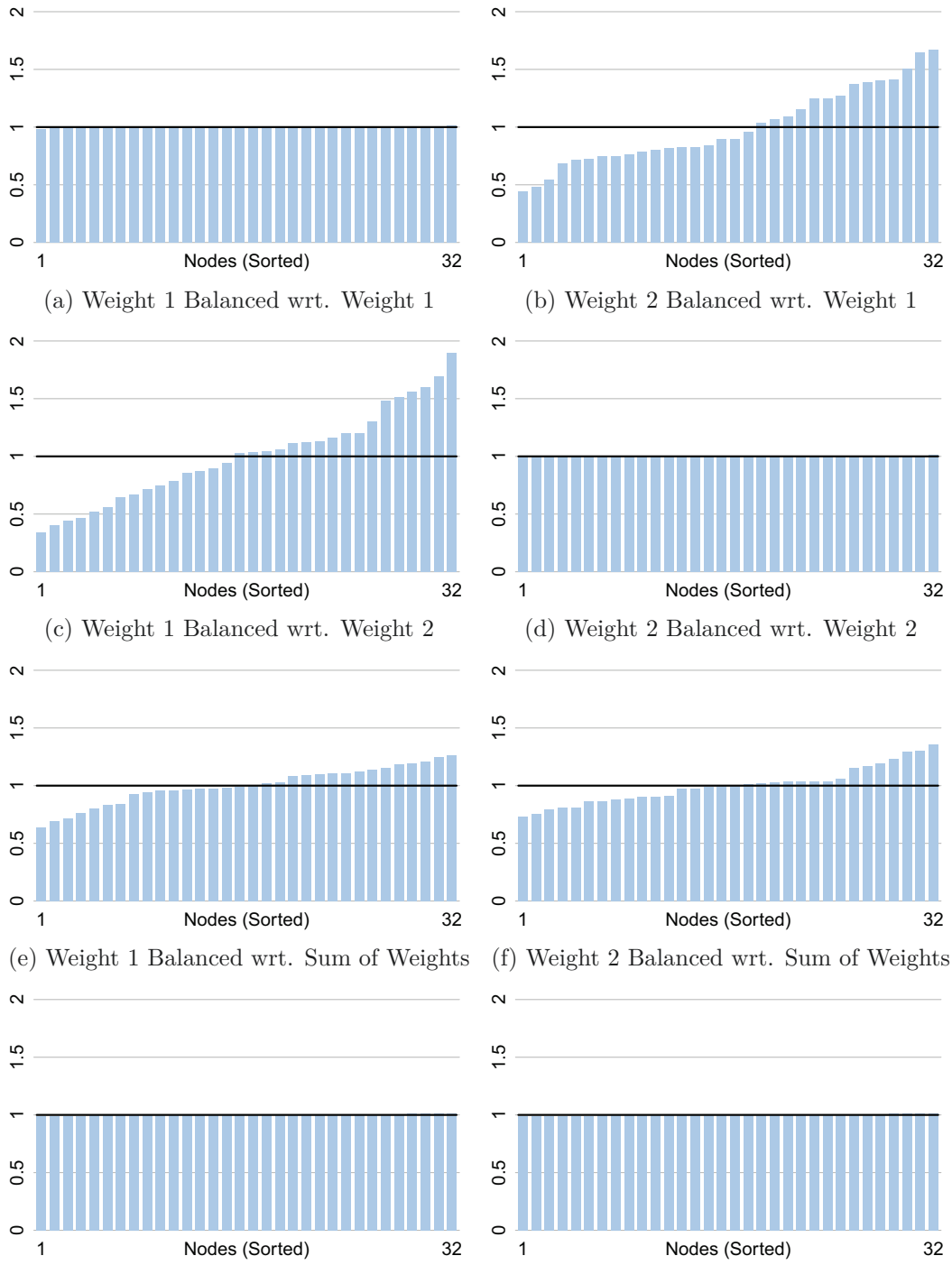
4.5.2. Synthetic Multi-Constraint Partitioning Experiment

In this experiment, we analyze the potential of the multi-constraint graph partitioning algorithm to generate partitionings that are superior to partitionings generated by the standard (single-weight) graph partitioning algorithm.

To run the experiments, we generate a synthetic workload graph that contains 1,000 vertices with two weights each. The vertex weights independently follow a Zipf distribution with the exponent $s = 1$. Thereby, we simulate a workload with a few large tasks and many small tasks. Comparable workloads can for instance be found in database-as-a-service systems (e.g., Schaffner et al., 2013). Each vertex in the workload graph is connected to a random number of neighbors. The number of neighbors is between 0 and 10 and again Zipf distributed ($s = 3$). The distribution of vertex weights (shown for a single weight) and neighbor counts in the workload graph are summarized in Figure 4.5. The resulting workload graph has 350 edges and an average vertex degree of 0.7.

All results of this experiment are shown in Figure 4.6. The graphs show eight relative node utilizations for four different partitionings and the two vertex weights. The first two graphs (Figures 4.6a and 4.6b) show the relative node utilizations regarding the first weight and the second weight when both are partitioned (and hence balanced) only using the first weight. Consequently, all partitions are balanced in the first weight but highly skewed in the second weight. Similarly, balancing the partitions with respect to the second weight leads to skew in the first weight (Figures 4.6c and 4.6d). When both weights are combined and then balanced, the resulting partitioning leads to skew in both weights (Figures 4.6e and 4.6f). Only when both weights are balanced individually in the multi-constraint graph partitioning algorithm are all nodes equally utilized (Figures 4.6g and 4.6h).

The experiments show that multi-constraint graph partitioning is able to individually balance multiple vertex weights while other strategies that rely on single-weight graph partitioning lead to skew in either one or both weights.



(a) Weight 1 Balanced wrt. Weight 1

(b) Weight 2 Balanced wrt. Weight 1

(c) Weight 1 Balanced wrt. Weight 2

(d) Weight 2 Balanced wrt. Weight 2

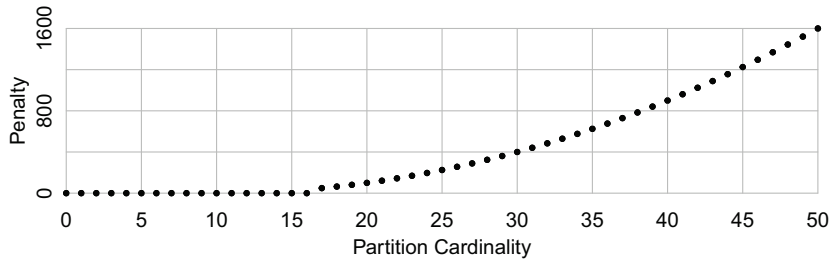
(e) Weight 1 Balanced wrt. Sum of Weights

(f) Weight 2 Balanced wrt. Sum of Weights

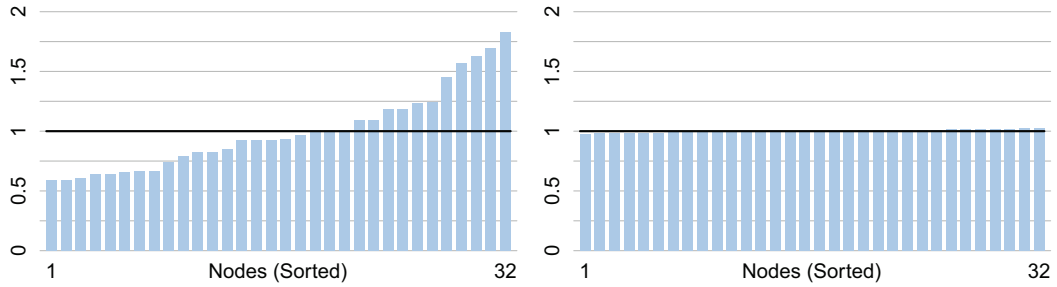
(g) Weight 1 Balanced Individually

(h) Weight 2 Balanced Individually

Figure 4.6.: Relative Node Utilizations in the Multi-Constraint Partitioning Experiment (Node Utilization Normalized to the Average Node Utilization)



(a) Exponential Penalty Function



(b) Node Utilization Using METIS

(c) Node Utilization Using PenMETIS

Figure 4.7.: Partitioning Experiment 1 using an Exponential Penalty Function (Node Utilization Normalized to the Average Node Utilization)

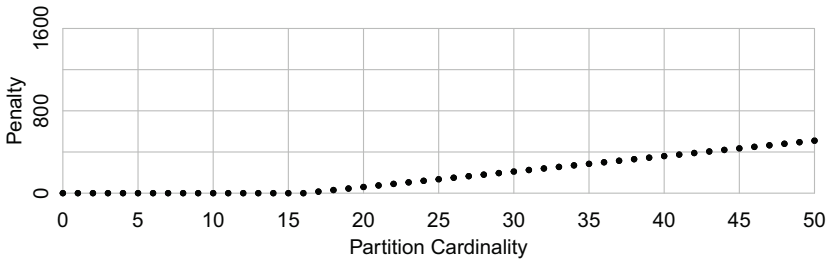
4.5.3. Synthetic Penalized Partitioning Experiment

To analyze the potential of penalized graph partitioning to balance load in presence of non-linear resources, we ran two synthetic partitioning experiments. The experiments use a synthetic workload graph and two different penalty functions. The motivation of these experiments is to compare partition weights (i.e., node utilization) computed by the unmodified graph partitioning algorithms in METIS to those computed by the penalized graph partitioning algorithm in PenMETIS.

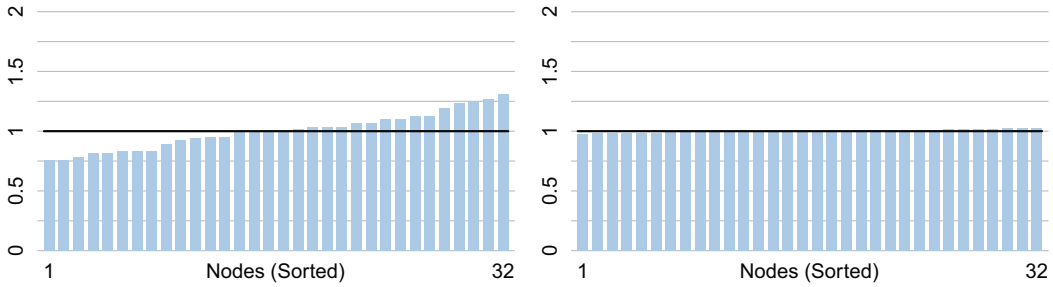
To run this experiment, we re-use the workload graph from the previous experiment (we use just the first vertex weight of each vertex). Recall that the workload graph contains 1,000 vertices with vertex weights that follow a Zipf distribution. Furthermore, each vertex has a random number of neighbors. The workload graph in this experiment is partitioned into 32 balanced partitions using METIS. Afterward, to estimate the actual load for each node, the penalty function is applied to each partition based on the partition cardinality. The resulting partition weights are compared to a second partitioning of the workload graph that was generated by the penalized graph partitioning algorithms in PenMETIS.

Exponential Penalty Function

In the first experiment, we assume an exponential penalty function. Thereby, we assume that the underlying resources can execute 16 parallel tasks, i.e., there is no penalty for smaller partitions. Starting with a cardinality of 17, the penalty grows with the square of the cardinality. The penalty function together with the node



(a) Linear Penalty Function



(b) Node Utilization Using METIS

(c) Node Utilization Using PenMETIS

Figure 4.8.: Partitioning Experiment 2 using a Linear Penalty Function (Node Utilization Normalized to the Average Node Utilization)

utilizations resulting from both partitioning methods are shown in Figure 4.7. The unmodified partitioning algorithm, which is unaware of the penalty, tries to balance the partition weights. The resulting relative weights (i.e., utilization) after applying the penalty are shown in Figure 4.7b. The node with the highest utilization receives 3.1 times the load of the node with the lowest utilization. The performance of this setup is bound to be suboptimal. In contrast, Figure 4.7c shows the utilization of all nodes based on the partitioning computed by the penalized partitioning algorithm. Here, all partitions are balanced within a tolerance of 3%.

Linear Penalty Function

In the second experiment, shown in Figure 4.8, we use a more conservative assumption for the penalty function. Here, the penalty grows linearly with the cardinality for partitions with 17 or more tasks. The resulting penalty function is shown in Figure 4.8a. The node utilization in both setups, using the unmodified and the penalized graph partitioning, are shown in Figures 4.8b and 4.8c. Even with the more conservative linear penalty function, partitioning the load without respecting the penalty leads to a difference in utilization of factor 1.7 between the most and least utilized nodes.

4.5.4. Scalability Experiments

In this section, we evaluate the overhead that penalized partition weights introduce in the partitioning process. Furthermore, we investigate how penalized graph parti-

tioning scales with the size of the graph, the number of partitions, and the number of constraints.

Walshaw Benchmark

We use example graphs from the Walshaw Benchmark (Soper et al., 2004) to analyze the overhead and scalability of penalized graph partitioning. Created in 2000 and maintained by Chris Walshaw, the Graph Partitioning Archive[7] is hosted at the School of Computing & Mathematical Sciences at the University of Greenwich. The archive contains 34 graphs together with their best known partitionings into different numbers of partitions (2, 4, 8, 16, 32, and 64) and with different imbalance parameters (0%, 1%, 3%, and 5%). New algorithms can be tested with the example graphs and better partitionings (regardless of the runtime) can be submitted to the archive. Currently, solutions of over 40 algorithms have been submitted, which makes the Walshaw Benchmark the most popular graph partitioning benchmark in the literature (Buluç et al., 2013). The example graphs in the archive are real world graphs from applications such as finite element computation, matrix computation, VLSI design, and shortest path computations. The graphs together with statistics on size and vertex degree are listed in Table 4.2. The graphs in the Walshaw Benchmark are per se no workload graphs of data-oriented systems. However, the performance of the penalized graph partitioning algorithms on these standard graphs yields valuable insights and indicates the capabilities of the method.

Penalized Partitioning Overhead

In this experiment, we investigate the overhead of penalized partition weights. Therefore, we first use the unmodified METIS to partition all example graphs into different numbers of partitions (2^1 to 2^{10}). Then, we use PenMETIS with penalized partition weights to partition the same graphs. The penalty function in this experiment is a simple constant function that adds a penalty of 1 to each partition. Regardless of the simplicity of the penalty function, the modified code path is used and balancing the penalized partition weights introduces an overhead. Figure 4.9 shows the absolute partitioning time in milliseconds needed for the unmodified graph partitioning and the penalized graph partitioning for each example graph.[8] The figure shows execution times for a partitioning into 64 partitions with an imbalance parameter of 3% (i.e., the heaviest partition may be 3% heavier than the average partition weight). The partitioning times for the other parameter values are similar in nature, i.e., the absolute partitioning times differ but only a small overhead for penalized partitioning can be seen.

As shown in Figure 4.9, partitioning the benchmark graphs takes less than a second in all but one cases. The largest graph (`auto`) contains 448,695 vertices and 3,314,611 edges and can be considered large in the context of workload graphs. Even this graph can be partitioned in about 1.4 seconds. The figure furthermore shows that penalized partition weights introduce a small overhead in the partitioning

[7] `http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/`

[8] We use a fairly moderate AMD Opteron (Istanbul) CPU running at 2.6 GHz for this experiment. As mentioned before, PenMETIS runs single-threaded.

	Size		Vertex Degree		
Graph	V	E	Min	Max	Avg
144	144,649	1,074,393	4	26	14.9
3elt	4,720	13,722	3	9	5.8
4elt	15,606	45,878	3	10	5.9
598a	110,971	741,934	5	26	13.4
add20	2,395	7,462	1	123	6.2
add32	4,960	9,462	1	31	3.8
auto	448,695	3,314,611	4	37	14.8
bcsstk29	13,992	302,748	4	70	43.3
bcsstk30	28,924	1,007,284	3	218	69.7
bcsstk31	35,588	572,914	1	188	32.2
bcsstk32	44,609	985,046	1	215	44.2
bcsstk33	8,738	291,583	19	140	66.7
brack2	62,631	366,559	3	32	11.7
crack	10,240	30,380	3	9	5.9
cs4	22,499	43,858	2	4	3.9
cti	16,840	48,232	3	6	5.7
data	2,851	15,093	3	17	10.6
fe_4elt2	11,143	32,818	3	12	5.9
fe_body	45,087	163,734	0	28	7.3
fe_ocean	143,437	409,593	1	6	5.7
fe_pwt	36,519	144,794	0	15	7.9
fe_rotor	99,617	662,431	5	125	13.3
fe_sphere	16,386	49,152	4	6	6.0
fe_tooth	78,136	452,591	3	39	11.6
finan512	74,752	261,120	2	54	7.0
m14b	214,765	1,679,018	4	40	15.6
memplus	17,758	54,196	1	573	6.1
t60k	60,005	89,440	2	3	3.0
uk	4,824	6,837	1	3	2.8
vibrobox	12,328	165,250	8	120	26.8
wave	156,317	1,059,331	3	44	13.6
whitaker3	9,800	28,989	3	8	5.9
wing_nodal	10,937	75,488	5	28	13.8
wing	62,032	121,544	2	4	3.9

Table 4.2.: Walshaw Benchmark Graphs Overview

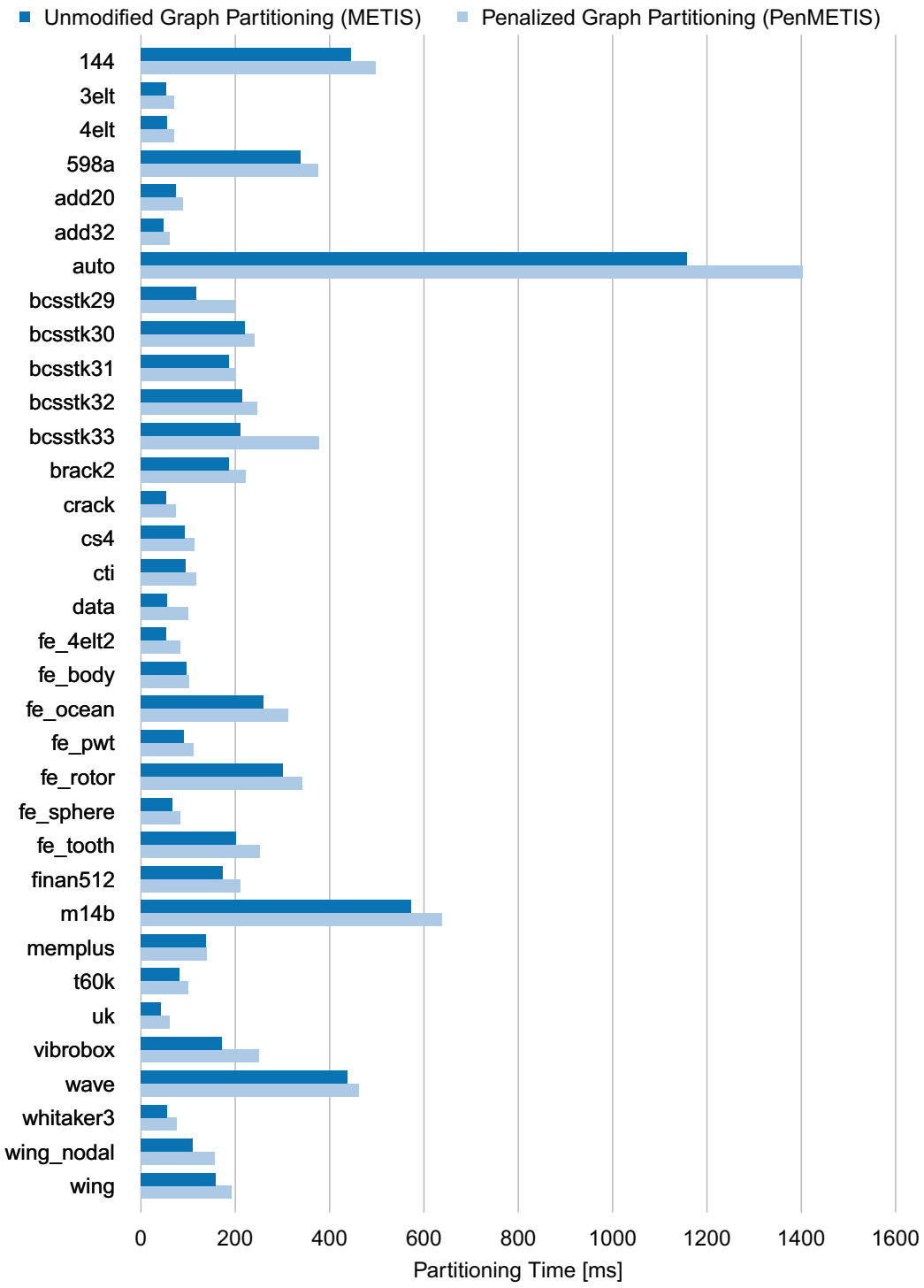
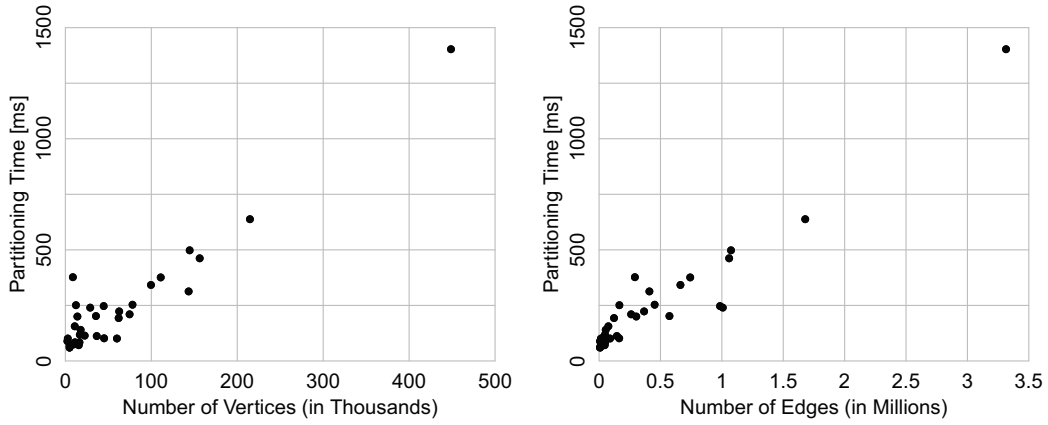


Figure 4.9.: Partitioning Time Comparison between Unmodified and Penalized Graph Partitioning (64 Partitions, 3% Imbalance)



(a) Scalability with the Number of Vertices (b) Scalability with the Number of Edges

Figure 4.10.: Execution Times of Penalized Graph Partitioning Charted by the Number of Vertices and Edges (64 Partitions, 3% Imbalance)

process that is usually negligible. On average, penalized graph partitioning takes 28% (42 ms) more time than the unmodified graph partitioning. The overhead is caused by the need to maintain vertex and partition cardinalities and the total partition weight. Furthermore, the frequent operation of moving a vertex from one partition to another partition is slightly more computationally expensive with penalized partition weights.[9]

Besides partitioning times, the comparison of unmodified and penalized graph partitioning also yields that PenMETIS achieves comparable total edge cuts and imbalance factors such that all balance constraints are met. A comprehensive overview of all experiment results is shown in Tables A.1 and A.2 in Appendix A.

Scalability with Graph Size

To use the allocation strategies based on graph partitioning with ever growing workload graphs, it is mandatory that the graph partitioning methods scale well with the size of the graph. Since penalized partition weights only induce a small overhead to the partitioning algorithm, the scalability of penalized graph partitioning is bound to the scalability of the underlying multilevel graph partitioning algorithms in METIS. Figure 4.10 shows the execution times of the penalized graph partitioning charted by the number of vertices (Figure 4.10a) and by the number of edges (Figure 4.10b). The charts indicate that the graph partitioning algorithms scale linearly with the number of vertices and the number of edges.

Scalability with Partition Count

In a second scalability experiment, we investigate how penalized graph partitioning scales with the number of partitions. Again, the scalability of the penalized

[9] Given that our implementation of penalized graph partitioning is only a proof of concept, there is room for code optimization that will bring the partitioning times closer to the unmodified graph partitioning.

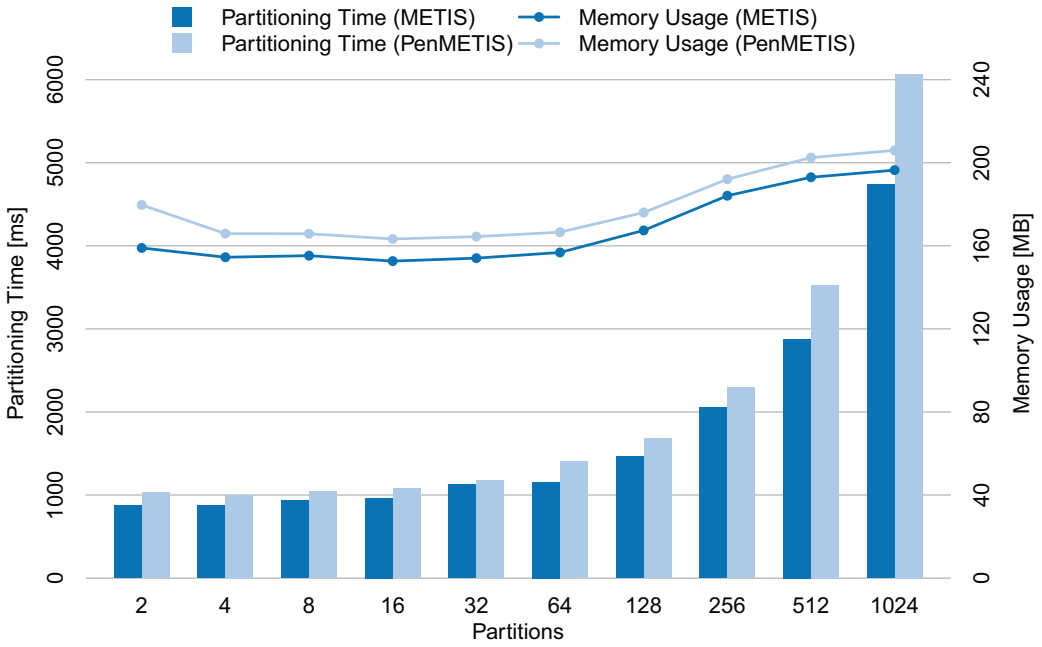


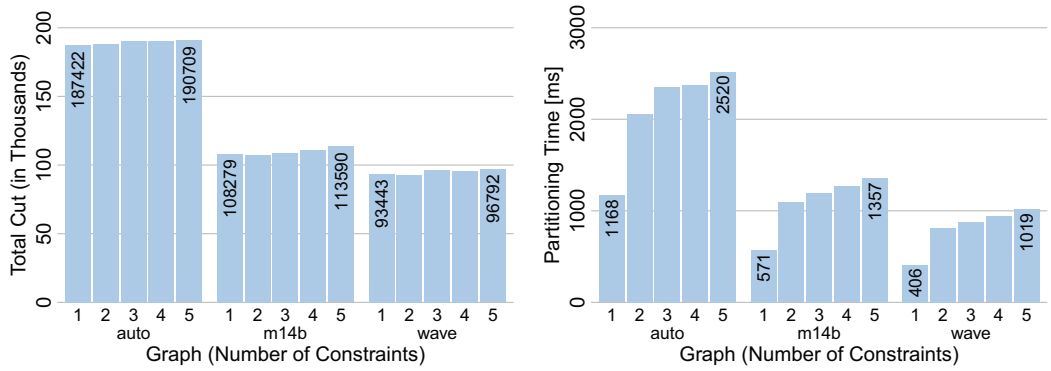
Figure 4.11.: Scalability of Graph Partitioning with the Number of Partitions (Graph `auto`)

partitioning method is closely related to the scalability of the underlying multilevel graph partitioning algorithm in METIS. For this experiment, we selected the largest benchmark graph (`auto`). However, results from other graphs are similar (all partitioning times are listed in Tables A.3 and A.4 in Appendix A). In Figure 4.11, we show partitioning times and memory usage for the unmodified graph partitioning (METIS) and the penalized graph partitioning (PenMETIS) for various partition counts. For small partition counts up 32, there is no increase in partitioning time or memory usage. The fixed parts (i.e., independent of the partition count) of the partitioning algorithms dominate the resource usage and execution time. Beyond 64 partitions, the partitioning time scales linearly with the number of partitions. The memory usage also increases beyond 64 partitions but seems to reach a plateau at about 200 MiB. With linear scalability in graph size and partition count, the balanced k-way min-cut allocation strategy is well suited to solve today's and future allocation problems.

Scalability with the Number of Constraints

In the last scalability experiment, we analyze the quality trend and scalability of the graph partitioning algorithm under different numbers of constraints. Our infrastructure model contains possibly multiple independent resources that are individually balanced in the multi-constraint graph partitioning problem. Hence, in this experiment, we evaluate the ability of allocation strategies based on graph partitioning to handle multiple resources.

For this experiment, we augment the graphs of the Walshaw Benchmark with



(a) Partitioning Quality (b) Partitioning Time

Figure 4.12.: Scalability of Graph Partitioning with the Number of Constraints (64 Partitions, 3% Imbalance)

vertex weights. Since the original graphs in the benchmark only have unit weights, we generate random weights. For each graph, we generate versions with 1 to 5 vertex weights per vertex. The weights are independently distributed uniformly between 1 and 100.[10]

To evaluate graph partitioning with different numbers of constraints, we partition each of the generated graphs into 64 partitions, measure the partitioning time and report the total cuts as quality measure. The graph partitioning algorithm was in all cases able to fulfill the balance constraints of 3% for all individual constraints. The results of our measurements are shown in Figure 4.12. Here, we present the results for three of the larger benchmark graphs in the Wahlshaw Benchmark, namely `auto`, `m14b`, and `wave`. However, all partitioning times and total cuts are listed in Tables A.5 and A.6 in Appendix A. Figure 4.12a shows that the total cut slightly increases with the number of constraints. This is to be expected since the algorithm has to fulfill more constraints, which in turn leaves fewer degrees of freedom to optimize the total cut. At the same time, partitioning time increases with the number of constraints (Figure 4.12b). This can be explained by the more complex routines to find matchings of vertices in the coarsening phase as well as by the increased number of balance steps in the uncoarsening phase.

Both, the decrease in quality and the increase in partitioning time are within small boundaries and we consider the graph partitioning method applicable to infrastructure and workload graphs with multiple resources. We conduct our experiments with up to five resources and we expect that adding resources will eventually cause the partitioning algorithm to fail in meeting a balance constraint. However, modeling considerably more than five resources seems unlikely in the targeted data-oriented systems.

[10] We also experimented with Zipf distributed weights and the results are similar.

4.6. Summary

In this chapter, we presented the balanced k-way min-cut graph partitioning problem. We showed how to solve the graph partitioning problem for single vertex weights (GPP) and multiple vertex weights, i.e., multiple resources in the infrastructure (MC-GPP). To reflect the non-linear behavior in our infrastructure model, we proposed new modifications of the graph partitioning problem (P-GPP and SW-GPP) to balance non-linear partition weights.

An evaluation of the different methods in synthetic experiments showed their potential to find balanced allocations in presence of multiple and non-linear resources. We furthermore showed in this chapter, that the graph partition algorithms scale well with the number of vertices, edges, partitions, and constraints and are hence applicable to a wide range of today's and future data-oriented systems.

The graph partitioning problem and its modifications presented in this chapter can be used as allocation strategies for a class of data-oriented systems. For systems with homogeneous nodes and a homogeneous network, an allocation is given by the partitioning of the workload graph together with an arbitrary mapping of partitions to infrastructure nodes.

Methods to further relax the assumptions made in this chapter, i.e., allocation strategies for heterogeneous infrastructures, are presented in the next chapter. In Chapter 5, we furthermore propose methods to incrementally update an allocation and to incorporate bounded resources in the graph partitioning problem.

5. Extensions to Graph Partitioning

The algorithms for the graph partitioning problems in the previous Chapter 4 solve the allocation problem for a class of data-oriented systems with static workloads and homogeneous infrastructures. Furthermore, the partitioning algorithms always balance resources, i.e., they assume unbounded resources. In this chapter, we present existing and propose new extensions and modifications to the basic algorithms from the previous chapter to relax the restrictions and to ensure that our allocation strategies fulfill all requirements collected in Section 3.4.

Incremental updates to the graph partitioning after changes in the workload or the infrastructure are covered in Section 5.1. In the subsequent Section 5.2, we discuss the challenges for the allocation strategies related to heterogeneous infrastructures. In Section 5.3, we present a method to incorporate capacity constraints, which are used to implement bounded resources, in graph partitioning algorithms. All our extensions are evaluated individually in synthetic experiments in Section 5.4.

5.1. Incrementally Updating the Graph Partitioning

Whenever the workload graph or the infrastructure changes, the partitioning needs to be re-evaluated. Changes in the workload graph can either be weights that change or vertices that are added or removed. These changes can be caused by merging or splitting vertices or by adding tasks to or removing tasks from the workload. Though not the focus of this thesis, we also consider changes in the infrastructure. Future systems may use adaptive hardware configurations to achieve performance goals or power consumption requirements (Fettweis et al., 2012). Changes in the infrastructure graph may manifest in changing vertex weights or changing edges. Communication links may be added or removed or the bandwidth may change. A special case needs to be considered when the infrastructure is dynamic: nodes that are removed from the system. On the one hand, the data stored on these nodes needs to be moved to other nodes before the node is deactivated (e.g., by setting vertex weights to zero first, which implicitly models no resource capacities on the node). On the other hand, the new infrastructure may not be able to accommodate all tasks which results in the allocation strategy not finding a valid solution.

Changes in the workload or the infrastructure may lead to an invalid partitioning, to partitions that are not balanced anymore, or to a suboptimal total cut. In this section, we describe strategies to incrementally update the partitioning to overcome these problems.

Once implemented, a partitioning of the workload graph is mapped to the infrastructure, i.e., each partition of the workload graph is mapped to a node in the infrastructure graph. Every vertex that is moved between partitions to compensate for the changes in the graph induces migration costs. Therefore, the new partition-

ing should be similar to the current partitioning to minimize the migration costs. 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. Optimizing for both goals simultaneously makes the partitioning problem a multi-objective optimization problem. Instead, a combination of both goals is commonly used in practice (Buluç et al., 2013).

5.1.1. Incremental Update Strategies

Here, we present and discuss basic incremental update strategies and heuristics that combine them. A more detailed discussion of the topic can be found in related articles on graph partitioning (e.g., Schloegel et al., 1997, 2000).

Local Refinement Strategy

One way to incrementally update a partitioning is to use the local refinement strategies already implemented in the multilevel graph partitioning framework. First, each vertex in the graph is assigned to its current partition. New vertices are assigned to the same partition as one of their randomly chosen neighbors. Then, the KL/FM method is used repeatedly to try to balance and refine the partitioning. Thereby, vertices from overloaded partitions are moved to underloaded ones. Depending on the extent of changes, local refinement strategies may not be able to fix the partitioning. If successful, refinement strategies commonly leads to good migration costs but the partitioning, i.e., the total cut, can deteriorate over time (e.g., Buluç et al., 2013).

Complete Repartitioning Strategy

A second method to account for graph changes is to partition from scratch. Here, the graph is partitioned without any knowledge of the previous partitioning. The new solution is then mapped to the previous one such that the migration costs are minimized. For instance, a greedy strategy maps heavy vertices first, i.e., vertices that would cause the highest migration costs. This complete repartitioning strategy commonly leads to good total cuts but high migration costs which may outweigh the benefit of the new partitioning (e.g., Buluç et al., 2013).

Virtual Vertices Strategy

A third strategy to incrementally update a partitioning is to introduce virtual vertices. A vertex with no weight that cannot be moved is added to each partition. Every other vertex is connected to the virtual vertex in the same partition. The edge weight on the edge that connects a vertex to the virtual vertex reflects the migration costs of this vertex. When the new graph, including the virtual vertices, is partitioned, the migration costs are accounted for in the total cut and therefore minimized by the partitioning algorithm (Hendrickson et al., 1996; Walshaw, 2010).

5.1.2. Update Cost Considerations

Deciding on whether or not or when to update the partitioning is an optimization problem by itself that depends on many details of the system and the application. Unless the current partitioning is rendered invalid by the changes in the workload or the infrastructure (e.g., by overloading a node's bounded resources or violating a balance constraint), the allocation strategy needs to decide whether updating the partitioning is beneficial. This decision is a tradeoff between the gain of the new partitioning in terms of better balance or lower total cut and the costs of migrating data to achieve the new partitioning.

Migration costs depend on the implementation of the data-oriented system and have, e.g., be investigated for whole virtual machines, that are used in private OS database-as-a-service systems (Rybina and Dargie, 2013; Rybina et al., 2014; Strunk, 2012). The costs can generally be quantified as the amount of data that needs to be moved. Schaffner et al. (2011) furthermore analyze the impact on other tasks' performances when data is migrated to or from a node. However, given the abstract infrastructure model in this thesis, it is hard to generally quantify the benefit of one partitioning over another partitioning. The assumption is that minimal communication and balanced unbounded resources lead to optimal performance. How much performance degrades when communication is added or partitions are not balanced cannot be predicted from the model.

Another aspect to consider is the frequency of workload changes. When the workload changes infrequently, migration costs will amortize eventually. With frequent workload changes, the decision on when to perform migrations is even harder.

Summary

We propose a hybrid strategy to incrementally update the graph partitioning. Whenever the graph changes such that the partitioning becomes invalid or the balancing constraint is violated, balancing and refinement steps based on the KL/FM method try to move vertices such that the partitioning is valid again. If no valid 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. To prevent the total cut in the graph to slowly deteriorate, every few local refinement operations, a new partitioning is computed in the background (even when the partitioning is still valid). The new partitioning replaces the current one only if the new total cut is considerably lower than the current total cut.

To address the challenging tradeoffs of migration, we propose to experimentally analyze the gain of migrations in the actual system and to use heuristics to decide when to migrate tasks.

5.2. Mapping Graph Partitions to Heterogeneous Infrastructures

When used as an allocation strategy, the GPP assumes a homogeneous infrastructure where each partition can equally well be assigned to any node. Precisely, the

GPP assumes homogeneous nodes in the infrastructure and a communication network in the shape of a complete graph with homogeneous links. However, our infrastructure model allows heterogeneous nodes with different capacities as well as heterogeneous communication networks. Heterogeneous nodes and different network connections between servers and between racks are often found in data centers that host database-as-a-service systems. Today's multiprocessor systems with NUMA characteristics commonly have homogeneous nodes, but a heterogeneous communication network (an overview and analysis of different NUMA systems is presented in Section 7.1). However, the trend towards heterogeneous processing elements like CPUs and GPUs in single server systems supports the need for allocation strategies that consider heterogeneous nodes (e.g., Borkar and Chien, 2011). In the presence of a heterogeneous infrastructure, finding an optimal mapping from partitions to infrastructure nodes becomes part of the GPP.

To describe algorithms that partition a workload graph and map it to a heterogeneous infrastructure, we need two additional definitions. For simplicity, we assume a single unbounded resource but all definitions naturally extend to multiple resources and bounded resources (bounded resources are covered in Section 5.3).

Definition 5.1 (Infrastructure Graph)**.** Given is an infrastructure as in the infrastructure model introduced in Section 3.1 (Definition 3.1). Let $I = (N, L, w_N, w_L)$ be the undirected, weighted infrastructure graph. The vertices N ($|N| = k$) in the graph are nodes of the infrastructure with capacities given by the node capacity function w_N. The edges L are communication links between nodes and the link capacities are expressed in w_L as per-unit communication costs. The node capacity and link cost functions are naturally extended to sets of nodes and links:

$$w_N(N') := \sum_{n \in N'} w_N(n) \text{ for } N' \subseteq N \text{ and}$$
$$w_L(L') := \sum_{l \in L'} w_L(l) \text{ for } L' \subseteq L.$$

Furthermore, let ρ be the routing table of the infrastructure, i.e., a mapping from each pair of nodes to a set of links that connect the nodes:

$$\rho\colon (N \times N) \to \mathcal{P}(L).$$

Definition 5.2 (Mapping)**.** Given a partitioning Π, let π be a mapping from partitions to the infrastructure nodes.

$$\pi\colon \{V_1, \ldots, V_k\} \to N.$$

5.2.1. Heterogeneous Nodes

We first consider an infrastructure with heterogeneous nodes, but a homogeneous communication network. The homogeneous communication network implies that $w_L(\rho(N_i, N_j)) = 1$ for any given $N_i, N_j \in N$ ($i \neq j$).

The balance constraint of the GPP can be modified to account for different node capacities. A partitioning that fulfills the new balance constraint contains partitions

according to the node capacities. The relative capacities of the target partitions (i.e., nodes in the infrastructure) need to be available as input parameter to the graph partitioning algorithm. The balance constraint of the GPP is modified as follows.

Definition 5.3 (Heterogeneous Balance Constraint)**.** 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 such that:

$$\frac{1}{k} \sum_{i=1}^{k} c_i = 1.$$

Let μ be the average partition weight:

$$\mu := \frac{w_V(V)}{k}.$$

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.

Given an infrastructure I, the relative partition capacities can be derived from the infrastructure nodes:

$$c_i = \frac{k \cdot w_N(N_i)}{w_N(N)}, \text{ for } i = 1, \ldots, k.$$

Using the heterogeneous balance constraint in the partitioning algorithm leads to a partitioning $\Pi = (V_1, \ldots, V_k)$ and a mapping π with $\pi(V_i) = N_i$.

5.2.2. Heterogeneous Links and Sparse Networks

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 and should therefore be part of the optimization process. 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.

Different approaches have been proposed in the past to solve the GPP with heterogeneous communication networks (e.g., Buluç et al., 2013, give an overview). One strategy, e.g., by Walshaw and Cross (2001), is to incorporate the cost of communicating data between nodes directly into the objective function during the partitioning process. Given a mapping π, the communication cost w_c between partitions needs to incorporate the workload graph edge weight (i.e., the communication volume) and the communication cost per unit (i.e., the communication distance):

$$w_c(E_{ij}) := w_E(E_{ij}) \cdot w_L(\rho(\pi(V_i), \pi(V_j))).$$

This approach requires the knowledge and availability of the pairwise communication costs. Depending on the size of the infrastructure, the size of the cost matrix

which is proportional to the square of the number of nodes may be large. Furthermore, it may not be feasible to evaluate these pairwise communication costs. Using the multilevel graph partitioning framework, incorporating the communication costs only applies to direct k-way partitioning algorithms. Using recursive bisection, it is not obvious what the communication cost between the two partitions is since they do not represent actual nodes of the infrastructure.

To overcome the problem with bisection, Pellegrini (1994) simultaneously bisects the workload and the infrastructure graph in the SCOTCH mapping library. In each step of the *dual recursive bipartitioning* algorithm, the workload graph and the infrastructure graph are partitioned in two parts with minimal cut and each workload partition is assigned to one of the infrastructure partitions. The bisection is recursively repeated until only one node remains in the partition of the infrastructure graph. All remaining vertices of the workload graph are assigned to this node.

A different strategy, termed *predictor-corrector*, was proposed by Moulitsas and Karypis (2008). Their approach is to first partition the graph without considering the infrastructure. In a second step, the partitioning is modified (corrected) according to the network characteristics. The correction step uses greedy strategies based on the same principles as the KL/FM method.

A last set of strategies that we like to mention also partition the workload graph without considering the infrastructure. In an attempt to minimize communication costs, these strategies then try to find a communication-optimal mapping from workload graph partitions to infrastructure nodes. An implementation of this strategy was, for instance, proposed by Brandfass et al. (2013). The authors use a greedy heuristic and map the partition with the highest total communication cost with respect to the already mapped partitions onto the node with the smallest total distance.

5.2.3. Heterogeneous Nodes and Links

The graph partitioning and mapping problem becomes even more complicated in presence of heterogeneous nodes and a heterogeneous network. Any of the strategies introduced in the previous section can be modified to account for heterogeneous nodes as well. However, the smaller degree of freedom makes it harder to find solutions. If, e.g., the workload graph was partitioned according to the node capacities, a communication-optimal mapping can only assign partitions to nodes that have the appropriate capacity. Assuming that there are classes of nodes with identical capacities, there may still be a choice, but the algorithm is more restricted compared to the case with homogeneous nodes. Partitioning methods and refinement strategies can be modified to take the different node capacities and communication costs into account. However, local vertex swapping heuristics may not always find optimal solutions.

Summary

Given the restrictions in the fully heterogeneous case, we focus on systems that either have heterogeneous nodes (and a homogeneous network) or have a heterogeneous

network (but homogeneous nodes). For the latter case, we implement the predictor-corrector method inspired by Moulitsas and Karypis (2008). The modifications of the refinement strategies based on the KL/FM method fit with the modifications already required in the same methods to enable penalized and secondary weight graph partitioning.

5.3. Capacity Constraints

The infrastructure model distinguishes bounded and unbounded resources. Bounded resources on the one hand impose a strict limit on their usage. Unbounded resources on the other hand can be overcommited and are transparently multiplexed by the system. While performance improves when unbounded resources are balanced across partitions, we assume that performance does not change depending on the balance of bounded resources. However, even when not balanced, usage of bounded resources must stay below the capacity or otherwise the partitioning is invalid.

To model both kinds of resources, bounded and unbounded, we propose an extension to the GPP. We add a *capacity constraint* to the already existing *balance constraint*. The capacity constraint is mainly for modeling purposes and we will show that partitioning algorithms can use the existing balance constraint with a carefully chosen imbalance parameter to also enforce capacity constraints.

In addition to Definition 4.4 (Balance Constraint), a capacity constraint is defined as follows. For the sake of readability, we assume a single vertex weight. However, the definition naturally extends to multiple weights and multiple capacity constraints.

Definition 5.4 (Capacity Constraint)**.** 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

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

The *total capacity* is the sum of all maximum partition weights:

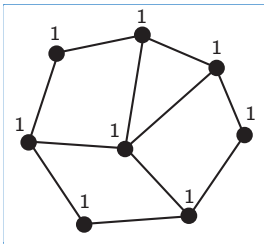
$$w_{\max}(V) := \sum_{i=1}^{k} w_{\max}(V_i).$$

The *total requirement* is the total partition weight of the graph w_Π.[1]

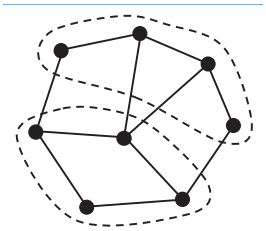
Given an infrastructure I, the maximum partition weights $w_{\max}$ are given in the node weights w_N.

Capacity constraints can be modeled with the already existing balance constraints and carefully selected imbalance parameters. Thereby, the existing solution heuristics for the GPP (also MC-GPP, P-GPP, and SW-GPP) can be used to partition graphs with balance and capacity constraints. The nature of capacity constraints

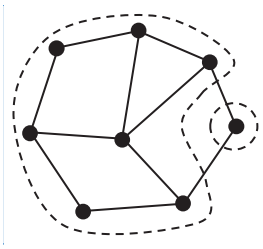
[1] Note that the total partition weight w_Π equals the total vertex weight w_V if partition weights are not penalized.



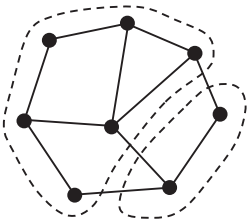
(a) Vertex Weights



(b) Valid Suboptimal Partitioning (Total Cut = 4)



(c) Invalid Partitioning ($w_V(V_1) > w_{\max}(V_1)$)



(d) Optimal Partitioning (Total Cut = 3)

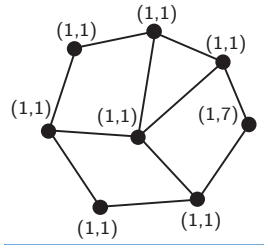
Figure 5.1.: Example of Graph Partitioning with Capacity Constraint

implies that the corresponding weights are usually not penalized. However, with the described solution, graphs with capacity constraints and penalized balance constraints can be partitioned.

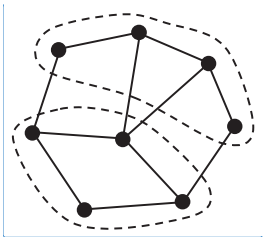
A naïve approach to enforcing a capacity constraint would be to use a balance constraint with a tight imbalance parameter ($\epsilon = 0$). As long as the total requirement is smaller than the total capacity, this leads to a valid solution (there is no solution otherwise). However, forcing tight balance severely reduces the solution space, which may lead to a suboptimal total cut. There may even be no valid solution at all.

Capacity Constraint Graph Example. Figure 5.1 shows an example graph where a too conservative imbalance parameter leads to a suboptimal partitioning. The vertex weights are shown in Figure 5.1a, edge weights are all equal to one. The maximum partition weights in the example are $w_{\max}(V_1) = w_{\max}(V_2) = 6$. Figure 5.1b shows a perfectly balanced partitioning ($\epsilon = 0$) with a total cut of 4 (not optimal). Figure 5.1c shows an invalid partitioning where the first partition contains more weight than the maximum weight. Figure 5.1d shows the optimal partitioning of the example graph with a total cut of 3. The capacity constraint is fulfilled although the partition weights are skewed ($w_V(V_1) = 6$, $w_V(V_2) = 2$).

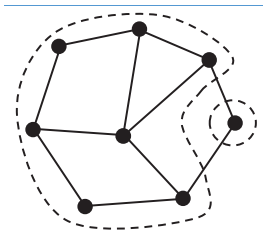
Capacity Constraint Graph Example (continued)**.** Figure 5.2 shows an example graph with a capacity constraint (first weight) and a balance constraint (second weight). The imbalance parameter for the balance constraint is $\epsilon_2 = 0.2$. The maximum partition weights are again $w_{\max}(V_1) = w_{\max}(V_2) = 6$. Figure 5.2b shows



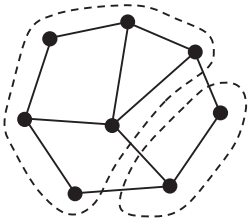
(a) Vertex Weights



(b) Invalid Partitioning (Balance Constraint)



(c) Invalid Partitioning (Capacity Constraint)



(d) Optimal Partitioning (Total Cut = 3)

Figure 5.2.: Example of Graph Partitioning with Capacity and Balance Constraint

an invalid partitioning. Although the first weight is perfectly balanced, the balance constraint of the second weight is violated. Figure 5.2c shows an invalid partitioning where the first partition contains more weight than the maximum weight. Figure 5.1d shows the optimal partitioning of the example graph with a total cut of 3 and the capacity and the balance constraint fulfilled.

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 resource requirements are considerably smaller than the total capacity and approaches 0 as the requirements approach the capacity.

Lemma 5.1. *In the case of homogeneous nodes, i.e,* $w_{max}(V_1) = \cdots = w_{max}(V_k)$*, a partitioning that fulfills a balance constraint with*

$$\epsilon = \frac{w_{max}(V)}{w_V(V)} - 1$$

also fulfills the capacity constraint.

Proof. The partitioning fulfills the balance constraint. Using the definitions, it fol-

lows that for all $i \in \{1, \ldots, k\}$:

$$\begin{aligned} w_V(V_i) &\leq (1+\epsilon) \cdot \mu \\ &\leq (1 + \frac{w_{\max}(V)}{w_V(V)} - 1) \cdot \mu \\ &\leq \frac{w_{\max}(V)}{w_V(V)} \cdot \frac{w_V(V)}{k} \\ &\leq \frac{w_{\max}(V)}{k} \end{aligned}$$

From $w_{\max}(V_1) = \cdots = w_{\max}(V_k)$, it follows that $w_{\max}(V) = k \cdot w_{\max}(V_i)$ for any given i. □

Capacity Constraint Graph Example (continued)**.** In the example graphs shown in Figures 5.1 and 5.2, the imbalance parameter ϵ needs to be set to $12/8 - 1 = 0.5$ (i.e., at most 50% imbalance) to enforce the capacity constraint without restricting the solution space.

In the case of heterogeneous nodes, the imbalance parameter can be chosen similarly to replace the capacity constraint with a heterogeneous balance constraint (see Definition 5.3).

Lemma 5.2. *In the case of heterogeneous nodes, let the normalized partition capacities be*

$$c_i := \frac{k \cdot w_{max}(V_i)}{w_{max}(V)}$$

A partitioning that fulfills a heterogeneous balance constraint with

$$\epsilon = \frac{w_{max}(V)}{w_V(V)} - 1$$

also fulfills the capacity constraint.

Proof. The partitioning fulfills the heterogeneous balance constraint. Using the definitions, it follows that $\forall i \in \{1, \ldots, k\}$:

$$\begin{aligned} w_V(V_i) &\leq (1+\epsilon) \cdot \mu \cdot c_i \\ &\leq (1 + \frac{w_{\max}(V)}{w_V(V)} - 1) \cdot \mu \cdot \frac{k \cdot w_{\max}(V_i)}{w_{\max}(V)} \\ &\leq \frac{w_{\max}(V)}{w_V(V)} \cdot \frac{w_V(V)}{k} \cdot \frac{k \cdot w_{\max}(V_i)}{w_{\max}(V)} \\ &\leq w_{\max}(V_i) \end{aligned}$$

□

To use capacity constraints in the GPP, either the total capacity $w_{\max}(V)$ (for homogeneous nodes) or the vector of partition capacities $(w_{\max}(V_1), \ldots, w_{\max}(V_k))$ needs to be given as an input parameter to the partitioning algorithm.

5.4. Experimental Evaluation

In this section, we experimentally evaluate the methods proposed in this chapter. The extensions for graph partitioning algorithms that enable incremental updates, heterogeneous infrastructures, and capacity constraints are tested with synthetic workloads.

5.4.1. Incremental Update Experiment

In the first experiment in this section, we evaluate the ability of our allocation strategies to react to changes in the workload. We therefore simulate a repeatedly changing workload and investigate the performance of our incremental update strategies described in Section 5.1.

We start our experiment with a random workload graph containing 1,000 vertices. The vertex weights follow a Zipf distribution between 1 and 100. Similar to the previous synthetic workload graphs (see Section 4.5.2), each vertex has a random number (between 0 and 10) of random neighbors. For this experiment, we also generate random edge weights (between 1 and 100) to get a more realistic evaluation of the total cut. The workload graph is initially partitioned into 32 partitions with an imbalance parameter of 3% (we use the same imbalance parameter throughout the experiment).

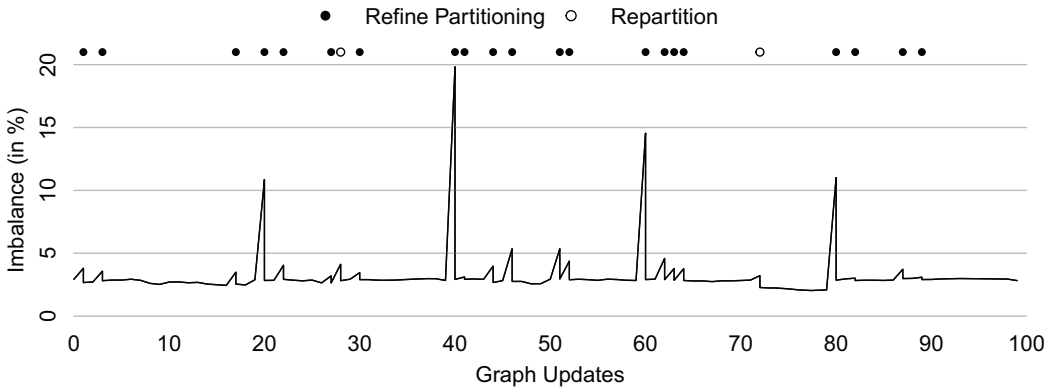
To simulate a changing workload, we define two workload graph modifications:

Minor Change A minor change is implemented by updating the vertex and edge weights of 1% of all vertices and all edges. The vertex weights are updated relative to their current weight, either increased or decreased by 10%. The edge weights are always increased by 20%.

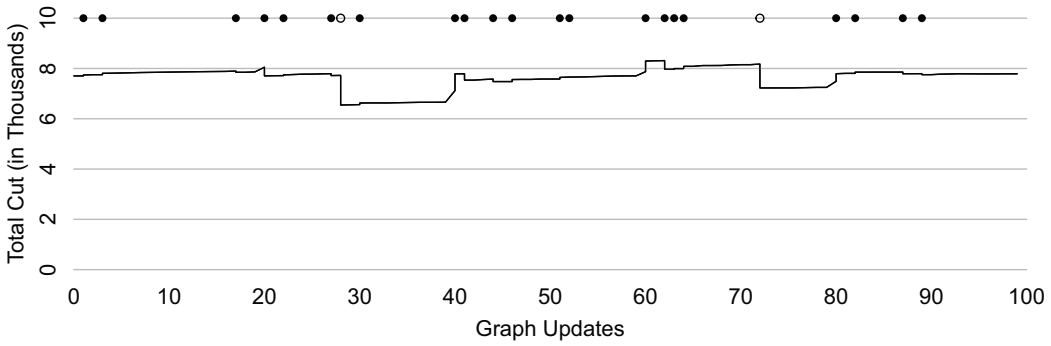
Major Change A major change is implemented by updating the vertex and edge weights of 10% of all vertices and all edges. The vertex weights are updated relative to their current weight, either increased or decreased by 80%. The edge weights of the randomly selected edges are doubled.

The complete experiment consists of 100 workload changes and the corresponding incremental updates to the partitioning. After 20 minor changes, one major change is simulated. The results of the experiment are shown in Figure 5.3.

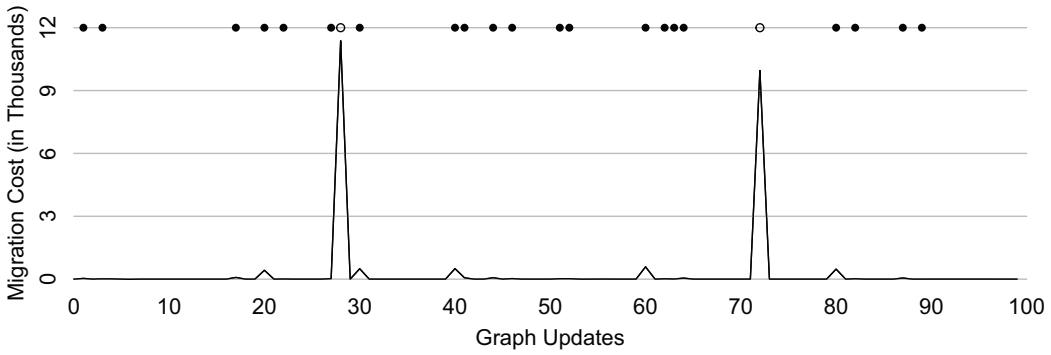
After each workload change, the current partitioning is evaluated against the new workload graph. The update mechanism is triggered when the evaluation yields that the balance constraint is violated, i.e., when there is more than 3% imbalance. The update strategy first tries to regain a balanced partitioning using local refinement strategies. A complete repartitioning is only triggered when the local refinement fails. In either case, the new partitioning is then evaluated against the workload graph and the imbalance as well as the total cut are logged. 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 total cut. The evolution of the graph imbalance and the total cut are summarized in Figures 5.3a and 5.3b. The results



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

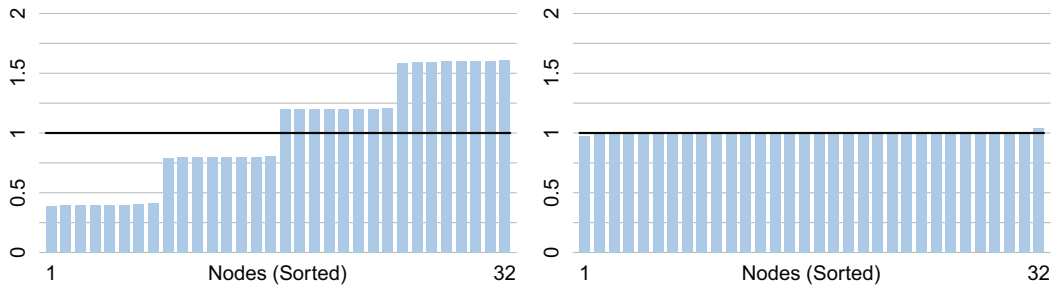


(b) Total Cut of the Graph in Thousands (Before and After Refinements/Repartitionings)



(c) Migration Cost for the Refinements/Repartitionings

Figure 5.3.: Incremental Update Experiment (32 Partitions, 3% Imbalance)



(a) Relative Partition Weights (Normalized to the Average Partition Weight)

(b) Relative Node Utilization Adjusted for Individual Node Capacities (Normalized to the Average Node Utilization)

Figure 5.4.: Partitioning Experiment with Heterogeneous Nodes (32 partitions, 3% Imbalance)

show that minor changes eventually and major changes always lead to violations of the balance constraint. However, in many cases (21 in the experiment) the local refinement algorithm is able to regain a balanced partitioning. Only in two cases, a complete repartitioning is triggered, which in both cases leads to considerably better total cuts.

Next to the imbalance and the total cut, the experiment reports the migration costs induced by the updates to the partitionings. 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 5.3c. The figure clearly shows that partitioning the workload graph from scratch causes considerably higher migration costs than refining an existing partitioning. The latter is true despite the fact that we implemented a greedy strategy to match previous partitions as good as possible to new partitions in an attempt to reduce the migration costs.[2]

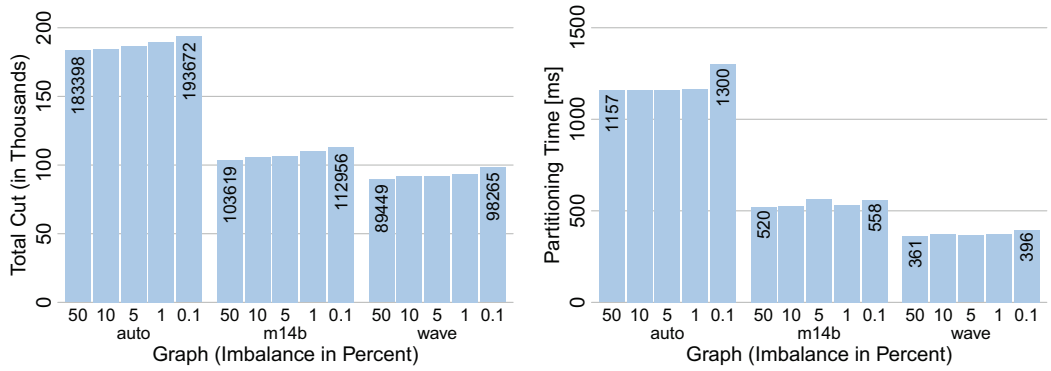
5.4.2. Heterogeneous Infrastructure Experiment

In the second experiment, we test the ability of our allocation strategy to consider heterogeneous nodes in the infrastructure. Heterogeneous nodes lead to imbalance in the system if all workload partitions are equal in size. Hence, to achieve a balanced system, the allocation strategy must incorporate the relative node capacities in the balance constraint.

We again use a synthetic workload graph with 1,000 vertices for this experiment. The vertices have a single weight that follows as Zipf distribution and a random number of neighbors (see Section 4.5.2).

To simulate a heterogeneous infrastructure, we subdivide the 32 nodes in the infrastructure in 4 classes with 8 nodes each. With the capacity of the fastest nodes as a baseline, the second class of nodes has three quarters of the baseline capacity,

[2]Migration costs for complete repartitionings are higher without this additional matching step. Our experiments showed cost reductions through the matching by as much as a factor of two in many cases.



(a) Partitioning Quality (b) Partitioning Time

Figure 5.5.: Ability of the Graph Partitioning Algorithm to Fulfill Capacity Constraints (64 Partitions)

the third class has half of the capacity, and the fourth class has only one quarter of the baseline capacity.

The workload graph is partitioned using PenMETIS and the relative node capacities are provided as an input parameter. Figure 5.4 shows the results of the experiment. Figure 5.4a on the left shows the relative partition weights yielded by the partitioning algorithm. The partitions are obviously skewed in size. However, as Figure 5.4b on the right shows, the node utilization is balanced across all nodes once the partition weights are adjusted for the node capacities.

5.4.3. Capacity Constraint Experiment

In the last set of experiments in this chapter, we evaluate the ability of our allocation strategy to fulfill capacity constraints. We showed in Section 5.3 that capacity constraints, though important for the concept, are implemented as balance constraints in the actual graph partitioning algorithm. The imbalance parameter needs to be carefully chosen to be restrictive enough to yield a valid partitioning without reducing the solution space. Hence, the experiments in this section evaluate the ability of PenMETIS to generate partitionings with different imbalance parameters.

For the capacity constraint experiments, we use the graphs in the Walshaw Benchmark. Since the vertices in the benchmark graphs have only unit weights, we add random vertex weights for the experiment. The vertex weights are uniformly distributed between 1 and 100. To evaluate the partitioning algorithm, we generate partitions with five different imbalance parameters, namely 50%, 10%, 5%, 1%, and 0.1%. We report the total cut as the quality measure and the partitioning time for each of the parameters. The results are summarized in Figure 5.5. Here, we present the results for three of the larger benchmark graphs in the Wahlshaw Benchmark, namely `auto`, `m14b`, and `wave`. However, all partitioning times and total cuts are listed in Tables A.7 and A.8 in Appendix A. Figure 5.5a on the left shows that the total cut increases with smaller imbalance parameters. This result is to be expected since the smaller imbalance parameter reduces the solution space and leaves the partitioning algorithm fewer options to balance. However, the increase in the

total cut from 50% imbalance to 0.1% imbalance is less than 10% in the worst of our experiments. Figure 5.5b on the right shows that the partitioning time only marginally increases for thighter imbalance parameters.

From an application point of view, it is more likely to have capacity constraints with a reasonable buffer which lead to imbalance parameters that are significantly higher than 0.1%. However, as the experiments show, our allocation strategies are able to fulfill even tight capacity constraints.

5.5. Summary

In the previous Chapter 4, we presented basic partitioning algorithms for variations of the graph partitioning problem. The strategies presented there can be used to partition graphs with single or multiple vertex weights, penalized vertex weights, and secondary vertex weights. In this chapter, we presented existing and proposed new modifications to the basic partition algorithms that (1) incorporate heterogeneous infrastructures, (2) react to changing workloads and infrastructures by incrementally updating a partitioning, and (3) enable bounded resources in the infrastructure model. Our experimental evaluation of the presented methods showed that they are able to overcome the challenges posed by the allocation problem.

The basic partitioning algorithms together with the proposed extensions fulfill all our requirements for an allocation strategy (see Section 3.4). Optimizing for communication costs is an inherent property of all min-cut partitioning algorithms. Non-linear performance is covered in the penalized and secondary weight graph partitioning algorithms. Multiple individual resources are enabled by the multi-constraint graph partitioning algorithm. Also recall that the two can be combined. Heterogeneous infrastructures are covered by the extensions in this chapter. Likewise, as shown in this chapter, bounded resources are respected in the allocation strategy. The last requirement, the ability to incrementally update an allocation, is fulfilled by the update strategies proposed in this chapter.

After the allocation strategies have been elaborated and evaluated in this chapter and the previous chapter, we present implementations of the strategies in actual data-oriented systems in the next two chapters. Unlike the experiments so far, which tested basic applicability and scalability of the methods, the next chapters contain experiments regarding end-to-end performance in two different systems. In Chapter 6, we implement and evaluate the allocation strategies in the database-as-a-service system MTM. Afterward, in Chapter 7, we do the same in the ERIS DBMS for multiprocessor systems.

6. Experimental Evaluation in the MTM Database-as-a-Service System

In this chapter, we experimentally evaluate our allocation strategies in a database-as-a-service system. The intention of the experiments is to implement and evaluate the allocation strategies presented in the previous chapters in a concrete system environment. The experiments are used to test whether the allocation strategies are applicable in an actual data-oriented system. Furthermore, the experiments help to evaluate the benefit of our new allocation strategies over existing strategies.

To implement and test the allocation strategies, we built a fully functional database-as-a-service system called *Multi-Tenancy Middleware* (MTM).[1] The MTM system is described in Section 6.1. To evaluate the performance of the database-as-a-service system under different allocation strategies, we need to generate, deploy, and query a large number of databases. As part of this thesis, we developed the multi-tenancy benchmark framework MulTe, which is described in Section 6.2. The setups, methodology, and results of our experiments are presented in Section 6.3.

6.1. Multi-Tenancy Middleware (MTM)

To evaluate our allocation strategies in a data-oriented system, we implemented a database-as-a-service system called *Multi-Tenancy-Middleware* (MTM). Our MTM system is in general similar to commercial database-as-a-service systems like Amazon RDS (Amazon, 2015) or Microsoft Azure (Microsoft, 2015). Databases can flexibly be provisioned on demand and the system takes transparently care of the physical representation of the databases. For our experiments, we use the MTM system with the ERIS in-memory database management system as backend (ERIS will be described in detail in Chapter 7). Here, we use a modified version of ERIS that is optimized towards running multiple instances in one server machine. The backend DBMS in MTM can be replaced with moderate effort (we did in fact implement the necessary components for MySQL and PostgreSQL as well). Our exemplary MTM system implements the *private process* class of database-as-a-service systems, i.e., each database provided by the system is backed by an instance of the ERIS DBMS containing a single database.

The MTM system has three major components, which are shown in Figure 6.1: (1) frontend tools, (2) a middleware, and (3) backend nodes. A sequence diagram for common database operations (i.e., create the database, connect to the database, and execute SQL statements) is shown in Figure 6.2. The actual database application

[1] Note that we use the terms *database-as-a-service system* and *multi-tenancy database management system* interchangeably.

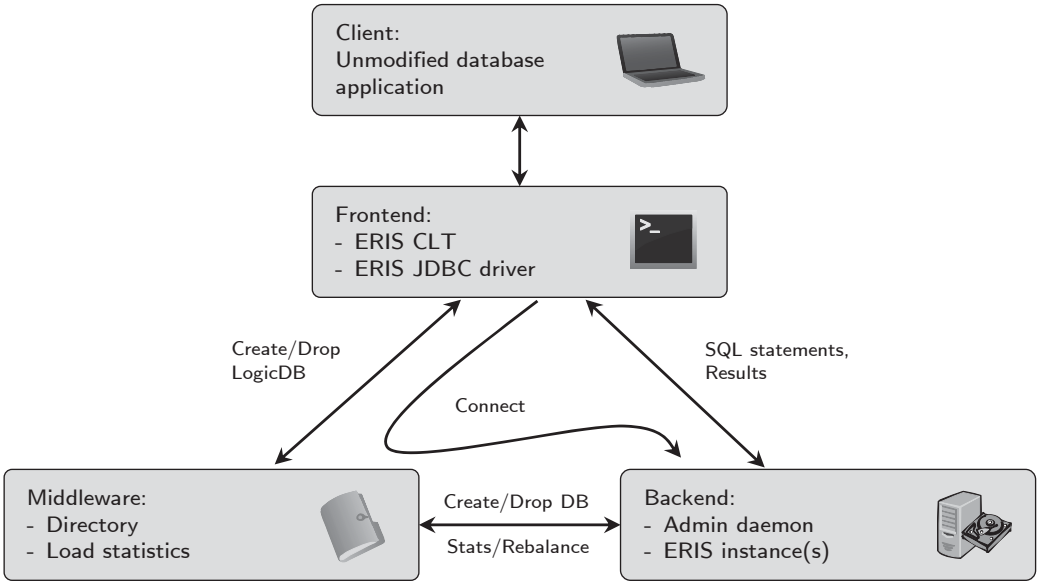


Figure 6.1.: System Architecture of the MTM System

does not have to be modified and can communicate transparently with the logical database provided by the MTM system.[2]

Frontend

The frontend tools are the interface between the client application and the middleware or backend node. Frontend tools in our exemplary system are modified versions of the ERIS Command-Line-Tool (CLT) and the ERIS JDBC driver. The modifications of the tools are to not directly connect to the database using the provided connection parameters (i.e., server IP address, port, and database). Instead, the frontend tools first communicate with the middleware using network protocols. Depending on the task, the frontend tools either delegate the task to the middleware (e.g., to create or drop a database) or request connection parameters for the actual backend node where the desired database resides (e.g., to execute SQL statements). The sequence of actions to create a logic database is shown in operation 1 in Figure 6.2. When the application tries to connect to the database, the frontend tools retrieve the connection information from the middleware and then directly and transparently connect to the correct backend node. Hence, the middleware is not involved in any subsequent communication between the application and the database and does therefore not impose a bottleneck. The sequences of actions to connect to the database and issue SQL statements is shown in the operations 2 and 3 in Figure 6.2.

[2] We refer to the database provided by the middleware as a *logical database*. Depending on the implementation class of the database-as-a-service system, this may be an actual database or just a schema in a database. The MTM system supports multiple modes, depending on the selected backend DBMS.

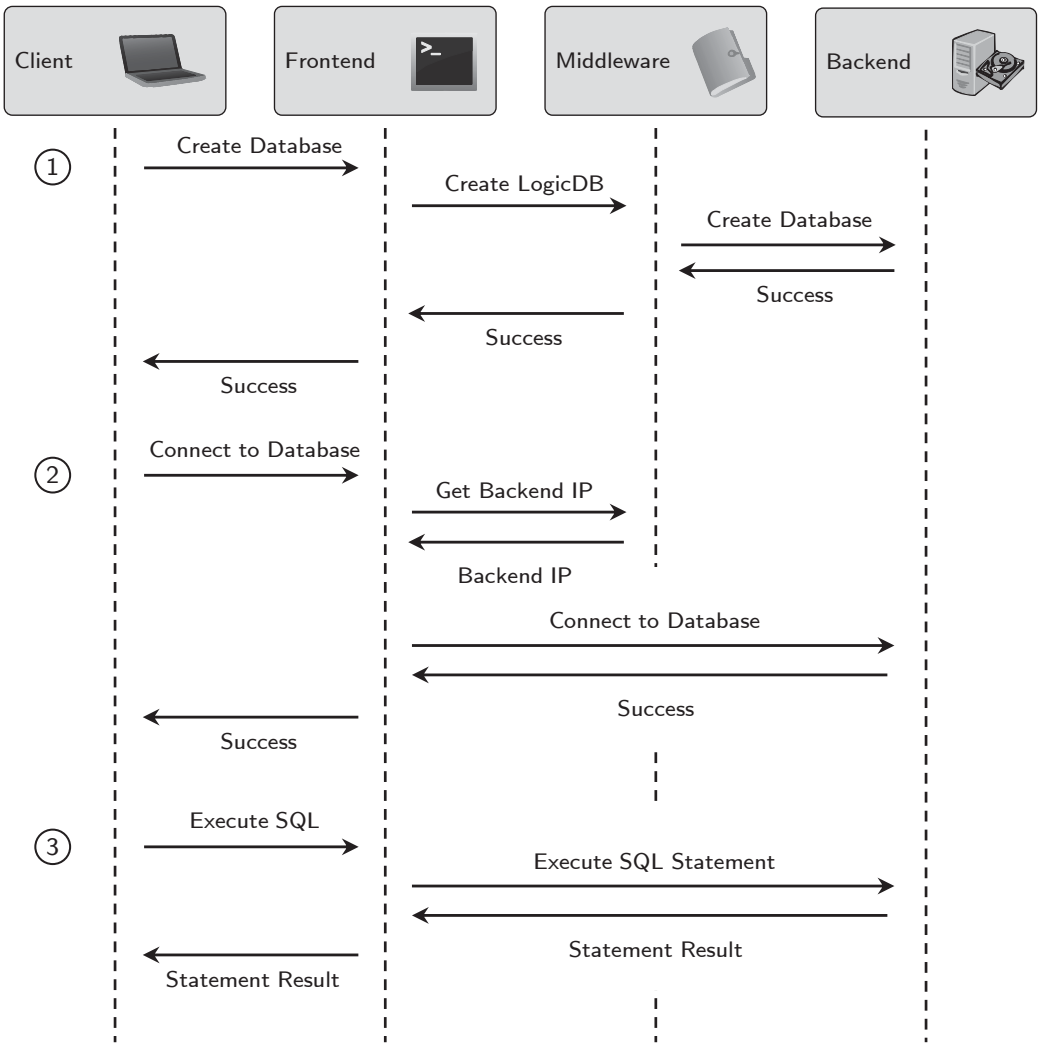


Figure 6.2.: Sequence Diagram for Basic Operations in the MTM System

Middleware

The middleware is the core component of the MTM system. The middleware maintains a directory of available backends (including connection information) and registered logical databases. When a frontend tool requests to create a logical database, the corresponding database (or schema) is created in the backend node and the logical database is registered in the directory. When a frontend tool wants to connect to a logical database, the connection parameters of the actual backend are returned. The middleware is also responsible for collecting load statistics and workload information as well as for implementing the allocation strategy. Based on the workload, the middleware assigns new logical databases to the appropriate backend node. Furthermore, the middleware issues migration tasks to backend nodes to balance load by moving databases between backends.[3]

Backend

The backend nodes in the MTM system host one or multiple instances of the backend database management system. In our exemplary system, we provide an instance of the ERIS DBMS for each logical database. Hence, our exemplary system is a private process implementation of a database-as-a-service system.[4] To start and stop ERIS instances on behalf of the middleware and to keep track of the used ports of the various instances, we use a lightweight administration daemon. This administration daemon also acts as distributed transaction coordinator in our performance experiments since ERIS does not support distributed transactions across multiple instances.

6.2. Multi-Tenancy Database Benchmark Framework (MulTe)

To evaluate the performance of the data-oriented system and to compare different allocation strategies, we need to generate, deploy, and query up to 1,000 databases. To enable our experiments, we developed the multi-tenancy benchmark framework MulTe as part of this thesis. We sketch the general architecture of the framework in this section while a more detailed description of the framework is provided in a dedicated paper (Kiefer et al., 2012). The framework itself is publicly available.[5]

6.2.1. Benchmark Framework Concepts

As shown in Figure 6.3, our approach in MulTe is to re-use existing benchmarks, including their schemas, data, and queries/statements. MulTe generates instances of these benchmarks to represent different tenants. Each tenant is given an individual, time-dependent workload that reflects a user's behavior. A workload driver runs all tenants' workloads against any multi-tenancy database management system and

[3] In our performance experiments, we provide the allocation from outside the MTM system using the benchmark tool to be able to easily compare different allocation strategies.
[4] We also implemented and experimented with a private database version of the MTM system.
[5] `https://wwwdb.inf.tu-dresden.de/research-projects/projects/multe/`

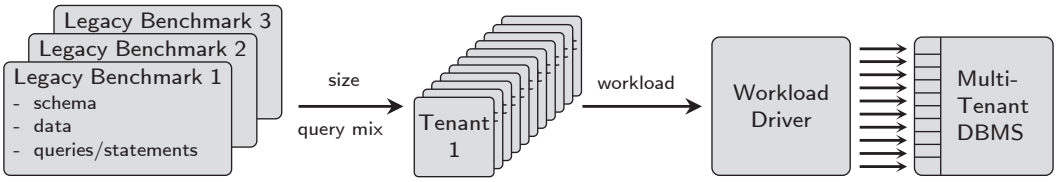


Figure 6.3.: MulTe Benchmark Framework Conception

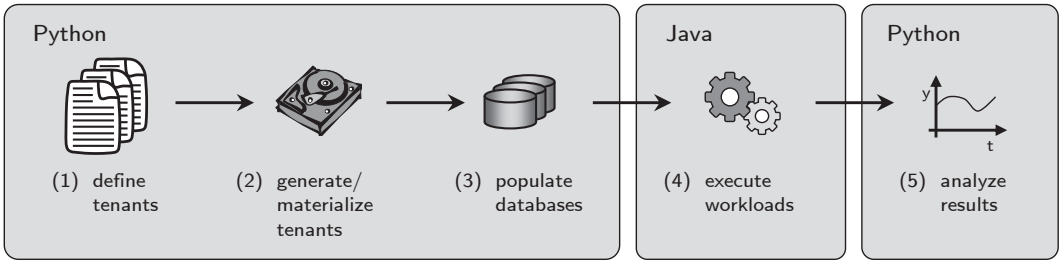


Figure 6.4.: MulTe Benchmark Framework Workflow

collects all execution statistics. All steps are supported with tools which together form the multi-tenancy benchmark framework MulTe.

The workflow of MulTe is shown in Figure 6.4. A set of Python scripts is used to define, generate, and populate tenants. The tenant definition contains the benchmark type (the Star Schema Benchmark in our experiments), the database size, the workload, and workload parameters such as the think time between query executions. In the second step, each tenant's database and workload are materialized using query and workload templates as well as the legacy data generator provided with the selected benchmark. In the third step, all databases are created and populated. Since MulTe operates against the MTM system in our experiments, a modified version of the ERIS command line tool is used that consults the middleware to create and populate databases (see previous Section 6.1 for details). Given the workload description, MulTe uses a Java workload driver to execute transactions in parallel and to collect statistics such as execution times (step 4 in Figure 6.4). In the last step, the results of a benchmark run can be summarized by MulTe. For the experiments presented in this chapter, we extended MulTe by additional scripts to compute the performance metrics that we are interested in.

6.2.2. Benchmark Framework Implementation

Our intention is to provide MulTe as a benchmark framework that will be used and extended by the community. We committed our work to a number of design principles that shall help to increase the value of the framework.

Easy-to-use: We provide example implementations to create tenants that run the TPC-H or the SSB benchmark against MySQL or PostgreSQL databases. A user only needs to provide a small number of configuration parameters (e.g., host, port, and paths) and a workload definition. The framework then generates and loads tenants. The workload driver can be used with a provided example configuration.

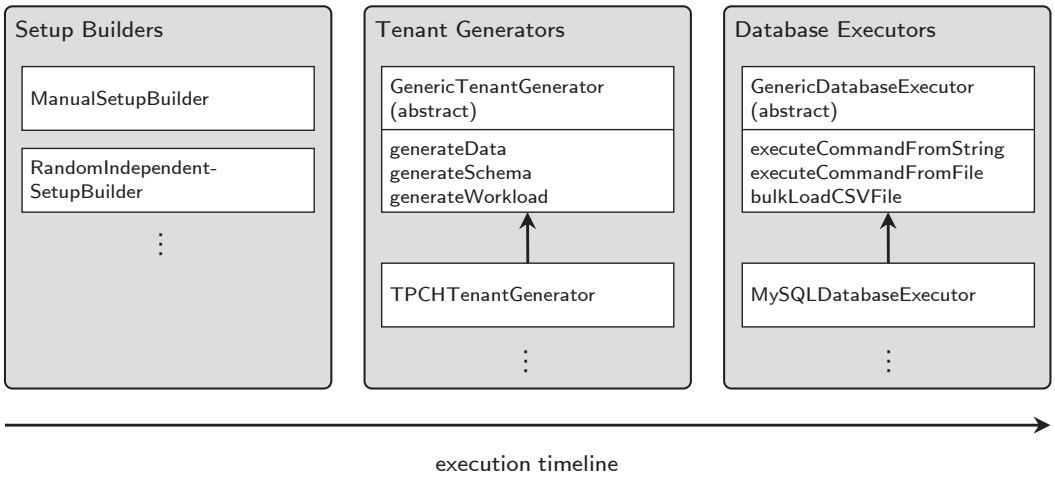


Figure 6.5.: MulTe Python Component Overview

Component Re-use: We re-use existing components wherever possible. From the given database benchmark, we re-use, e.g., data generators and schema definitions. The Java workload driver in MulTe is a modification of the workload driver of the TPoX database benchmark (TPoX, 2015).

Extensibility: The framework components to generate and load tenants can be replaced such that both, other benchmarks and other database management systems, can easily be supported.[6] The Python language allows users to quickly adopt our example implementations and modify them to fit their needs. The workload driver is designed to be able to run a wide variety of workloads, thus supports different benchmarks. Extending the workload driver to support different database management systems is also possible as long as they provide a JDBC interface.

Python Scripts—Define, Generate, and Load Tenants

The Python scripts/classes to define, generate, and load tenants follow the structure shown in Figure 6.5. The full Python functionality can be used to define a tenant's set of activity parameters. We provide two implementations that either specify the parameters' values explicitly or pick them randomly (following a distribution) but independent from one another. Given the tenants' definitions, instances of the respective database benchmarks are generated using both, provided data generators and templates. Other instance generators for other benchmark types can be added by implementing the methods `generateData`, `generateSchema`, and `generateWorkload`. A DBMS-specific database executor is used to load all tenants' data into the databases. Our example implementation uses the command line interfaces of MySQL and PostgreSQL to execute SQL statements and to bulk-load data from CSV files.

[6] We proved the extensibility of the framework by adding support for the MTM system to MulTe.

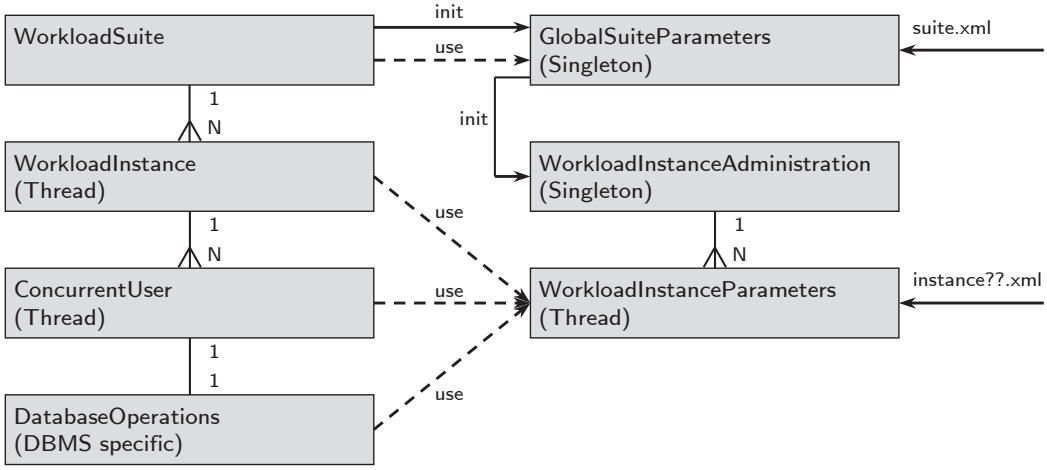


Figure 6.6.: MulTe Workload Driver (Architecture Modification Overview)

Java Workload Driver

The Java workload driver, which executes all tenants' workloads against a database-as-a-service system, is a modification and extension of the workload driver provided with the TPoX database benchmark. A detailed documentation of the original workload driver and its capabilities can be found on the TPoX website[7]. Here, we would like to only mention some of the modifications that we made. Figure 6.6 shows an overview of all changes. The class structure is closely coupled to the framework conception. The `WorkloadSuite` uses `GlobalSuiteParameters` for a benchmark, taken from a simple configuration file (here `suite.xml`). The `WorkloadSuite` furthermore drives multiple tenants (`WorkloadInstance`) that in turn use multiple `ConcurrentUsers` to execute statements in the database. All of a tenant's parameters—stored in `WorkloadInstanceParameters`—are taken from an XML configuration file that is automatically generated by MulTe.

6.3. System Performance Experiments

In the following experiments, we test our allocation strategies in the MTM database-as-a-service system. The goal of the experiments is to evaluate whether allocation strategies based on graph partitioning lead to better system balance (and hence worst-case performance) than a greedy baseline allocation strategy. Furthermore, the experiments test whether penalized graph partitioning outperforms the unmodified graph partitioning in our system.

6.3.1. Hardware Setup

We conduct our experiments using Amazon Web Services[8]. Multiple EC2 instances are used as backend nodes. Additional EC2 instances are used to host the benchmark

[7] `http://tpox.sourceforge.net/`
[8] `http://aws.amazon.com`

Setup	Frontend & Middleware	Backend Nodes	Database Count
SETUP40	1 x c4.2xlarge	4 x m4.xlarge	40
SETUP1K	4 x c4.2xlarge	32 x m4.xlarge	1,000

Table 6.1.: Performance Experiment Setups

Instance Type	vCPU	Memory (GiB)	Physical Processor	Clock Speed (GHz)
m4.xlarge	4	16	Intel Xeon E5-2676 v3 (Haswell)	2.4
c4.2xlarge	8	15	Intel Xeon E5-2666 v3 (Haswell)	2.9

Table 6.2.: Amazon EC2 Instance Details

application, frontend tools, and the middleware.[9]

We deploy two different setups, summarized in Tables 6.1 and 6.2, for our experiments. The first setup (SETUP40) consists of 4 instances as backend nodes (`m4.xlarge`) and a single instance to run the benchmark (`c4.2xlarge`). 40 databases in total are hosted in this first setup. In the second setup (SETUP1K), 32 instances are used as backend nodes and 4 instances drive the benchmark. In this second setup, we host a total of 1,000 databases with an aggregated size of 120 GiB.

6.3.2. Workload/Workload Model

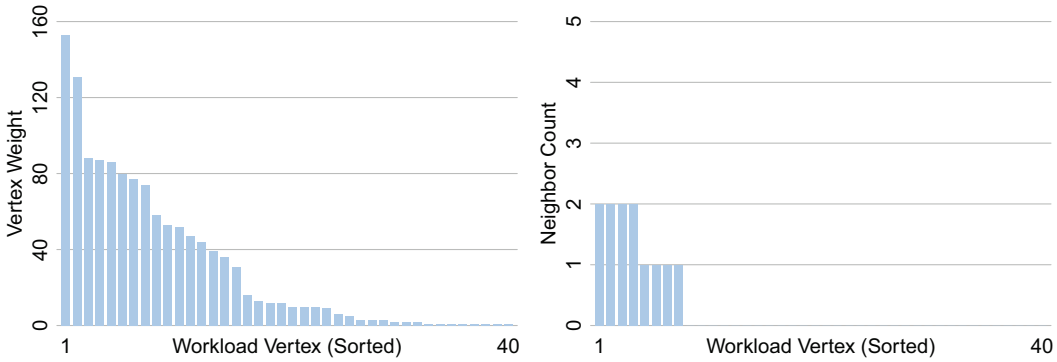
We use the Star Schema Benchmark (SSB) (O'Neil et al., 2009) as the basis to generate our experiment workloads.[10] Prior to the experiments, we evaluated the cost of each of the 13 queries in the SSB based on a cost model that uses the ERIS execution model and query execution plans generated by ERIS. For the experiments, we decided to model a single processing cost. 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 sizes times 13 queries in the benchmark[11]). Together with a think time between queries, we are able to generate databases with any given intensity/cost.

To run the experiments, we generate a workload graph with random vertex weights and random edges. The graph consists of either 40 or 1,000 vertices, depending on the setup. The vertex weights represent processing costs and follow a Zipf distribution with exponent $s = 1$ (Schaffner et al. (2013) observed similarly distributed tenants in an actual database-as-a-service system). Each vertex has a number of (randomly selected) neighbors. The number of neighbors also follows a Zipf distri-

[9] Frontend and middleware in the MTM system are separate components that communicate via network protocols and can hence be deployed on different machines. However, to reduce the total cost of the experiments (without compromising its results), we decided to run the benchmark tool, frontend tools, and the middleware on the same EC2 instance in our setup.

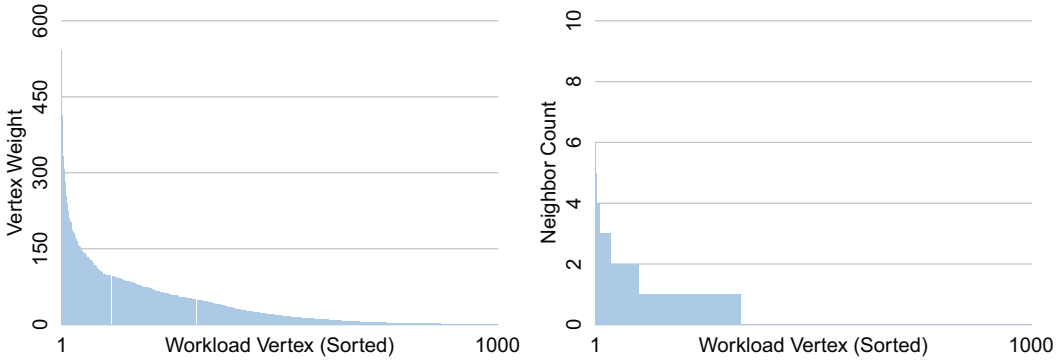
[10] One reason to use the Star Schema Benchmark is that all queries proposed in the benchmark can be executed in the ERIS prototype, although it does not yet support the full SQL standard.

[11] To keep the setup as simple as possible, each database runs only a single query type.



(a) Vertex Weight Distribution (b) Neighbor Count Distribution

Figure 6.7.: Workload Characteristics (Setup40)



(a) Vertex Weight Distribution (b) Neighbor Count Distribution

Figure 6.8.: Workload Characteristics (Setup1k)

bution between 0 and 6 with an average number of neighbors of 0.5. The distributions of vertex weights and neighbor counts for both workload graphs are shown in Figures 6.7 and 6.8.

Neighbors (or edges in the workload graph) can result in distributed transactions in the actual workload, depending on the allocation. For each vertex, a database (i.e., a database size and a query type) is selected that matches the vertex' weight best. A think time is added to match the database's processing cost exactly to the generated vertex weight. The database size is added as a second weight to the vertex. Consequently, our workload model contains two resources: (1) processing cost as an unbounded resource and (2) database size as a bounded resource. The resulting workload consists of a list of databases and the transactions that are executed on them (either single-sided or distributed).

6.3.3. Infrastructure Model

The infrastructure model in our experiments is based on the Amazon EC2 instances and the ERIS database management system. Following the workload, we model two resources in our infrastructure: processing capacity and memory capacity. Since

processing is an unbounded resource, only relative capacities are needed in the infrastructure model. Given that all backend 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 backend node has 16 GiB main memory which is added as the second (bounded) capacity to each node in the infrastructure graph. Due to the lack of detailed network information, we assume that all instances can communicate equally well with all other instances. Hence, the infrastructure graph in our experiments is a fully connected 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 using the EC2 instances and ERIS. As a result, for our penalized graph partitioning allocation strategy we model a 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 (i.e., running 5 workload vertices on one node results in a combined load that is equal to the 5 vertices plus an additional 6th small vertex).

6.3.4. Allocation Strategies

We compare three different allocation strategies in our experiments: (1) First Fit (FF), (2) Unmodified Graph Partitioning (UGP), and (3) Penalized Graph Partitioning (PGP).

Distributing the databases across all backend nodes using a round-robin strategy seems to be a simple (though naïve) 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. To avoid this problem, we use a greedy method (first fit) as a baseline allocation strategy instead. The First Fit method sorts all databases by their descending size and allocates each database to the least utilized backend node that is able to accommodate it. Using this strategy leads (in most cases) to valid allocations. Furthermore, using the least utilized backend in each step tries to balance the load across all nodes.

The second allocation strategy in our experiments (UGP) is based on the unmodified graph partitioning algorithm in METIS. Here, we partition the workload graph (without penalty) to get a balanced allocation with minimal communication.

The third allocation strategy (PGP) uses our penalized graph partitioning algorithm in PenMETIS. The penalty function of the infrastructure model is used to describe the infrastructure behavior and to produce a better balanced allocation with minimal communication.

6.3.5. Performance Metrics

To evaluate the performance of the database-as-a-service system in the various setups and hence to evaluate the quality of the allocation strategies, we introduce the Relative Response Time (RRT) as a performance metric and look at three statistical measures: (1) Average Relative Response Time (AvgRRT), (2) Median Relative

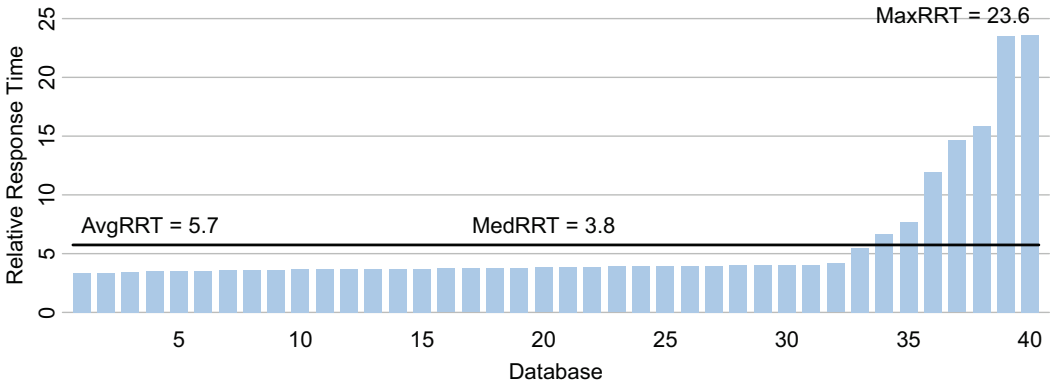
Response Time (MedRRT), and (3) Maximum Relative Response Time (MaxRRT). Computing the performance metrics is done in three steps. First, each absolute response time collected by the workload driver is transformed into a relative response time using a previously performed baseline run (i.e., the relative response time is relative to the best-case execution). Thereby, we are able to compare different transactions with different absolute response times. 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. The 95% quantile is also more robust to small numbers of outliers compared to, e.g., the maximum relative response time per database. Given one relative response time for each database, AvgRRT, MedRRT, and MaxRRT across all databases can be computed in the third step as the quality measures for the whole data-oriented system.

6.3.6. Experiment Results

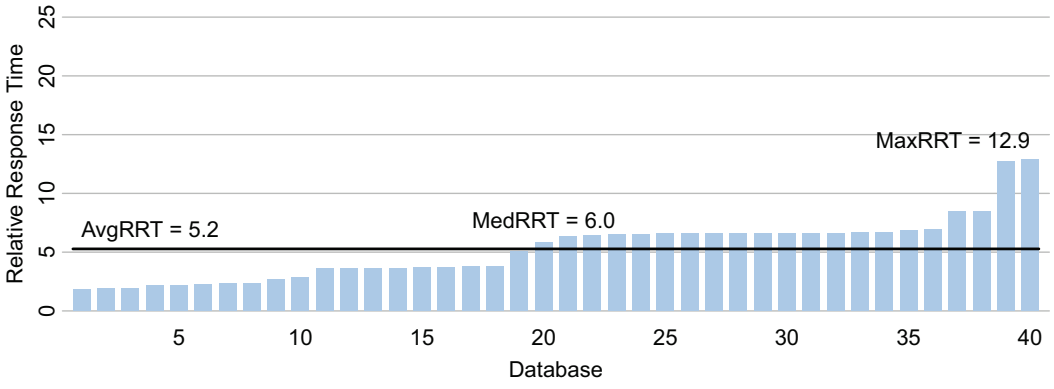
Having all the building blocks, the experiments are run using the following workflow. First, the setup is prepared based on the allocation strategy that is to be evaluated. To prepare the setup, a list of databases including the workload parameters is provided to the MulTe framework. Since the goal of our experiments is to compare different allocation strategies, the allocation (i.e., the backend node for each database) is also part of the MulTe input.[12] MulTe then creates and loads all databases in the MTM database-as-a-service system. Once created, a baseline run is conducted where each database workload is executed individually to get best-case response times for each database. This baseline is later used to compute relative response times for the performance metrics. After the baseline run, the actual experiments run for 30 minutes. In these experiment runs, the MulTe workload driver executes all workloads repeatedly and in parallel and collects execution statistics. The workload driver logs all executed transactions together with the respective response times. In a last step, we use different scripts to analyze the log of the experiment run and to compute the performance metrics.

The results of our experiments using Setup40 are summarized and visualized in Figure 6.9. Figures 6.9a to 6.9c show the relative response times for all 40 databases for the First Fit allocation strategy, the Unmodified Graph Partitioning allocation strategy, and the Penalized Graph Partitioning allocation strategy respectively. Figure 6.9a confirms that the First Fit strategy is unaware of the distributed transactions. While balancing most of the databases' response times in this experiment, FF leads to bad response times for some databases (MaxRRT=23.6). The allocation strategies based on graph partitioning are able to reduce communication which leads to better response times for distributed transactions, which are now co-located in most cases (MaxRRT=12.9 and 11.5 respectively). However, the Unmodified Graph Partitioning strategy only balances the vertex weights and fails to account for the infrastructure behavior. This leads to one node hosting 16 of the 40 databases.

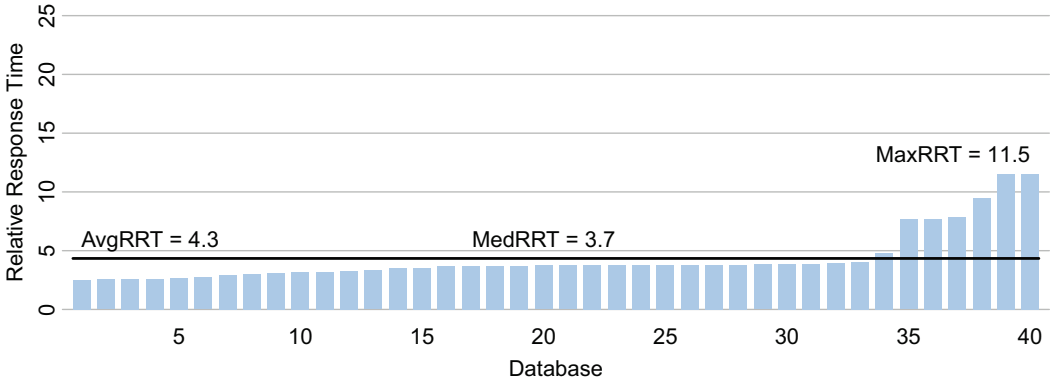
[12] In an actual database-as-a-service system, the middleware controls the allocation based on load statistics and the implemented allocation strategy. The allocation is then transparent to the user.



(a) First Fit Allocation Strategy (FF)



(b) Unmodified Graph Partitioning Allocation Strategy (UGP)



(c) Penalized Graph Partitioning Allocation Strategy (PGP)

Figure 6.9.: Relative Response Times, SETUP40

Consequently, the actual performance on this node is worse than predicted by the allocation strategy. The best overall system performance in all three performance metrics is achieved by the Penalized Graph Partitioning strategy that both reduces communication and balances load.

Given the small setup, the performance gain (MaxRRT and AvgRRT) can be considered good. However, as can be seen in Figure 6.9c, even PGP fails to perfectly balance the performance of all databases. We are not able to pinpoint the exact reasons for the remaining performance skew in this complex system setup. System behavior that we are unaware of or assumptions that we make in the workload model, the infrastructure model, or the allocation strategy may have led to the skew in response times shown in the figure.

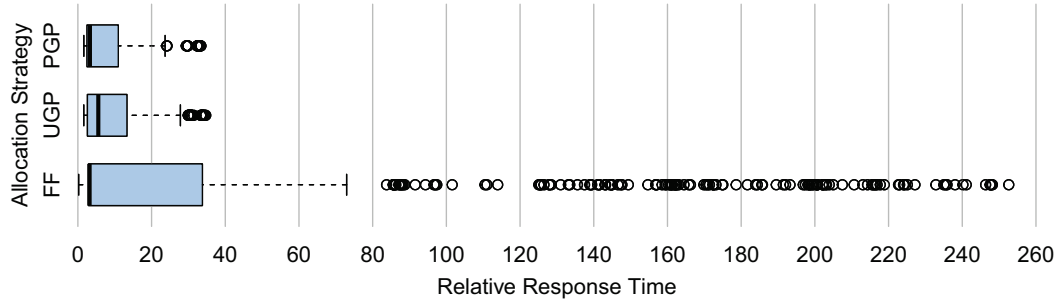
In the second setup, which contains 1,000 databases on 32 backend nodes, we investigate whether the allocation strategies are able to improve the balance in larger systems. The results of our experiments using Setup1k are summarized as box-whisker plots in Figure 6.10. Figure 6.10b shows the same results as Figure 6.10a on a differently scaled x-axis (UGP and PGP only). The plots in Figure 6.10a show that the First Fit allocation strategy fails to balance the load, which leads to many outliers and a MaxRRT of 252. Comparing the allocation strategies that are based on graph partitioning, it can be seen in Figure 6.10b that Penalized Graph Partitioning leads to better overall system performance in all metrics. PGP causes fewer outliers than UGP and has better maximum, average, and median relative response times. As a reference, computing the PGP allocation for the workload graph in Setup1k takes 6 ms using PenMETIS.

6.4. Summary

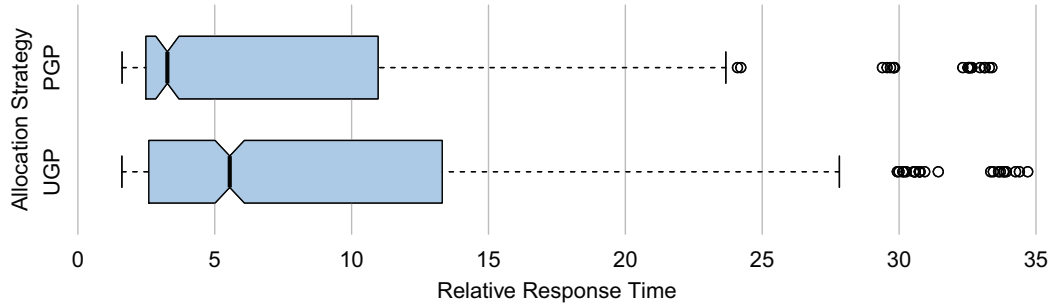
In this chapter, we presented MTM, our fully functional private-process database-as-a-service system. Based on a Java middleware and modified frontend connection tools, MTM is able to flexibly and transparently provide databases as a service to the user. The physical representation of the databases is implemented using the ERIS DBMS. Furthermore, we presented MulTe, our multi-tenancy database benchmark framework which can be used to generate, deploy, and query a large number of databases.

The experiments to test the end-to-end performance showed the applicability of our allocation strategies in an actual data-oriented system. Based on Amazon Web Services and the Star Schema Benchmark, we deployed and analyzed two setups with 40 and 1,000 databases respectively. The results demonstrated the benefits of the penalized graph partitioning allocation strategy over other allocation strategies.

In the next chapter, we continue our performance experiments in a different data-oriented system, namely the ERIS DBMS for multiprocessor systems.



(a) Relative Response Time Distribution, All Allocation Strategies



(b) Relative Response Time Distribution, Only Graph Partitioning Strategies

Figure 6.10.: Relative Response Time Distribution, SETUP1K (Note Different X-Axes)

7. Experimental Evaluation in the ERIS DBMS for Multiprocessor Systems

In this chapter, we evaluate our allocation strategies in the ERIS database management system. ERIS is an in-memory DBMS that is optimized for the challenges of multiprocessor systems, i.e., systems with multiple sockets and multiple cores. A defining characteristic of these systems is the non-uniform memory access (NUMA) caused by the main memory being directly attached to the various sockets and hence distributed across the system. Differences in latency and bandwidth up to a factor of 10 between local and remote memory accesses lead to the conclusion that NUMA systems should be treated as distributed systems. We show in this chapter that our allocation strategies can be used to optimize the mapping of data (containers) to the various NUMA nodes.

To gain a deeper understanding of the NUMA characteristics of modern multiprocessor systems, we conduct a series of low-level benchmarks on three different machines. The results as well as the conclusions from the experiments are presented in Section 7.1. The ERIS system, which is designed and built based on the findings of the low-level experiments, is described in Section 7.2. There, we present the system architecture and core components of the ERIS system. In Section 7.3, we detail our performance experiment with the full ERIS system and the allocation strategies developed in this thesis. The results of our experiments are presented towards the end of this chapter.

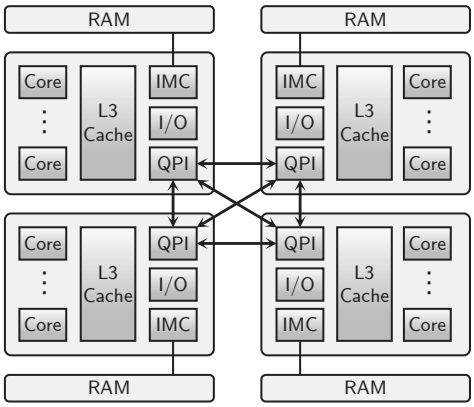
7.1. NUMA Characteristics in Multiprocessor Systems

In this section, we briefly introduce NUMA system architectures and present low-level–benchmark results of three different NUMA machines. The presented results yield important insights in designing scalable database management systems in the presence of non-uniform memory access and are fundamental for the ERIS DBMS.

As a reference, the tested NUMA systems and their main characteristics are depicted in Figure 7.1.

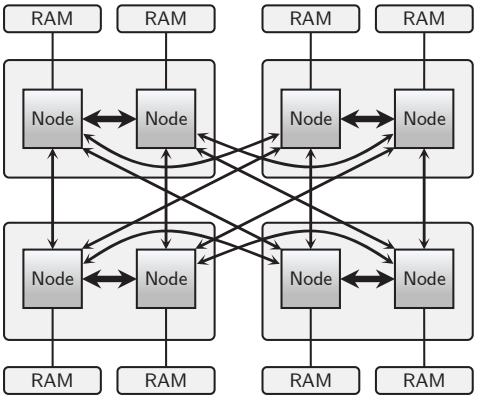
7.1.1. NUMA System Architecture

NUMA systems consist of multiple interconnected multiprocessors (also referred to as *nodes*), where each multiprocessor contains multiple processing units (cores) and an integrated memory controller (IMC). The multiprocessors can transparently access all memory locations, although the installed memory is distributed among the IMCs in different multiprocessors. Latency and bandwidth of memory accesses depend on the distance between the requesting multiprocessor (source node) and the



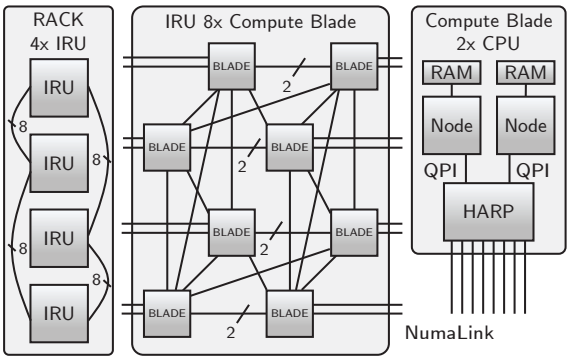
- 4x Intel Xeon E7-4869
- 40 cores (80 HW threads)
- 128 GiB memory (32 GiB per node)
- 24 MiB LLC per socket
- QPI: 12.8 GiB/s per link

(a) Intel Machine (Detailed View)



- 4x AMD Opteron 6274 (dual node)
- 64 cores
- 64 GiB memory (8 GiB per node)
- 12 MiB LLC per socket (6 MiB per node)
- HyperTransport: 12.8 GiB/s per link

(b) AMD Machine (Topology View)



- 64x Intel Xeon E5-4650L
- 512 cores
- 8 TiB memory (128 GiB per node)
- 20 MiB LLC per socket
- QPI: 16 GiB/s to HARP
- NumaLink6: 2x 6.7 GiB/s betw. HARPs

(c) SGI Machine (Topology View)

Figure 7.1.: NUMA Machines used for Evaluation

multiprocessor that contains the data (home node). The *local memory* associated with each multiprocessor is accessed with low latency at a high bandwidth. In contrast, *remote memory* is accessed via point-to-point connections (Hypertransport Technology Consortium, 2010; Intel, 2009) between the multiprocessors that add latency and limit the achievable bandwidth. In the worst of the tested cases the latency of a remote access is approximately 10 times higher and the bandwidth is limited to about 11% in comparison to local accesses.

Multiple levels of caches are commonly used to mitigate the performance impact of the above-mentioned latency and bandwidth constraints. The caches are distributed over the multiprocessors as well and all currently available NUMA systems enforce cache coherence to maintain a consistent view of all processing units on the shared address space. Hackenberg et al. (2009) showed that the overhead of the coherence protocol caused by accesses to shared data can be severe in large systems.

Naturally, data placement is an important aspect to consider with NUMA systems and data should be located close to the multiprocessor that accesses it frequently. The default data placement policy of Linux is called *first touch*. Newly allocated memory is placed local to the thread that actually writes (touches) it for the first time. It is, however, possible that memory is allocated on remote memories. Moreover, the default thread scheduler in Linux operating systems may migrate threads frequently to different multiprocessors, although it prefers intra-node thread migrations to inter-node migrations (Blagodurov and Fedorova, 2011). This leads to remote memory accesses, even when the memory was allocated locally in the first place. Hence, the operating system leaves many opportunities for suboptimal (i.e., remote) memory access patterns. This is especially true when many threads access a large portion of the main memory as it commonly happens in main memory database management systems.

7.1.2. Low-Level–Benchmark Results

For the low-level–performance experiments, we use three different NUMA systems ranging from 4 multiprocessors and 64 GiB of main memory to 64 multiprocessors and a total of 8 TiB of main memory. The hardware specifications of the *Intel machine* with 4 multiprocessors, the *AMD machine* with 8 multiprocessors (on 4 sockets), and the *SGI machine* with 64 multiprocessors are summarized in Figure 7.1. To gain deeper insights in the performance of the three different NUMA machines, we conduct several low-level benchmarks. The best-case bandwidth and latency are upper bounds for the achievable performance and help to reason about the performance of in-memory DBMSs. All measurements are performed with the BenchIT tool (Hackenberg et al., 2009). The results are shown in Table 7.1.

Intel Machine

The nodes of the Intel machine are fully connected via QPI links (Intel, 2009) as depicted in Figure 7.1a. The results of our experiments show that the latency of remote memory accesses is only 50% higher compared local accesses. The impact of the QPI link on the achievable bandwidth is more severe as it results in 2.5 times lower data rates. However, the effects of the non-uniform memory access

distance	bandwidth (GiB/s)	latency (ns)
local	26.7	129
1 hop QPI	10.7	193

(a) Results on the Intel Machine

distance (link width)	bandwidth (GiB/s)	latency (ns)
local	16.4	85
1 hop HT (full link)	5.8	136
1 hop HT (split,single)	4.2	152
1 hop HT (split,dual)	2.9	152
2 hops HT (split,single)	3.7	196
2 hops HT (split,dual)	1.8	196

(b) Results on the AMD Machine

distance	bandwidth (GiB/s)	latency (ns)
local	36.2	81
2nd processor	9.5	400
1 hop NUMALink	7.5	510
2 hops NUMALink	7.5	620
3 hops NUMALink	7.1	740
4 hops NUMALink	6.5	870

(c) Results on the SGI Machine

Table 7.1.: Low-Level–Benchmark Results of NUMA Machines

are small compared to the other two machines as communication between any two multiprocessors requires only one hop via QPI.

AMD Machine

The second machine in our setup is an AMD machine. As shown in Figure 7.1b, it is actually a 4-socket system where each socket houses a *dual node package*. The two nodes in a package communicate via HyperTransport (Hypertransport Technology Consortium, 2010) which practically results in a system with 8 multiprocessors. Each multiprocessor has 4 HyperTransport ports to connect to either the I/O subsystem or to other multiprocessors. As a unique feature of the AMD machine, HyperTransport links can be split into sublinks to connect a node with two other nodes with just one HyperTransport link. However, this results in different link bandwidths for different links. Additionally, even with split links, the AMD machine is not fully connected and certain routes require two hops.

As indicated in Figure 7.1b, the two nodes that share a socket are connected via a dedicated (not split) HyperTransport link and can therefore utilize the full 16-bit link width. Connections between other nodes are implemented with 8-bit sublinks and hence have a lower connection bandwidth. Furthermore, some of the split links

only have one sublink populated (denoted by *split,single* in Table 7.1) while both sublinks are occupied on other links (denoted by *split,dual*). Our experiments mirror these characteristics; depending on the distance of memory and accessing thread, we measure six different bandwidths and four different latencies. The disparities between local access and the furthest remote access are a factor of 9.1 in bandwidth and 2.3 in latency. Given the interesting properties of this system, it has been analyzed in a dedicated performance analysis paper. We refer to Molka et al. (2014) for additional in-depth performance results for the AMD machine.

SGI Machine

The third machine in the setup is an SGI UV 2000 with 64 multiprocessors and a total of 8 TiB main memory. An overview of the topology is shown in Figure 7.1c. The system consists of 1 rack that houses 4 Individual Rack Units (IRUs). Each IRU consists of 8 Compute Blades, that in turn contain 2 multiprocessors each. Each socket is equipped with an 8-core Intel Xeon CPU with 128 GiB of local main memory.

The two multiprocessors in a Compute Blade are connected via QPI to a communication hub called HARP. The HARPs are NumaLink hubs that connect the multiprocessors in a Compute Blade to other Compute Blades in the same as well as in other IRUs. As shown in Figure 7.1c, each blade in the system has 8 connections to other blades. Each connection consists of two NUMALink6 links, one for each multiprocessor in the blade. The 8 blades in an IRU are connected as a 3D enhanced hypercube (SGI, 2012). Each blade in an IRU is additionally connected to two blades in other IRUs. This topology leads to connections with up to four hops and six different bandwidths.

Measuring all possible distances reveals that the differences in bandwidth and latency between local access and the furthest remote access are as high as factor 5.5 and 10.7, respectively.

7.1.3. Design Principles for NUMA-Aware DBMSs

From the general NUMA architecture as well as our benchmark results, we derive that scalable in-memory database management systems must be designed for memory locality. On the one hand, explicitly reading or writing remote memory suffers from up to ten times higher latency and significantly lower bandwidths. On the other hand, remote and concurrent memory accesses lead to cache concurrency as well as worse cache locality and hence higher cache coherence overhead. As a conclusion, scalable database management systems for multiprocessor systems with non-uniform memory access must provide adaptive partitioning of the data. Moreover, by means of data and thread placement, the data management system must minimize remote memory accesses by primarily accessing local data objects. In turn, this leaves sufficient link capacities for remote accesses caused by inevitable communication during analytical query processing and by load balancing. The operating system cannot provide sufficient locality due to its insufficient knowledge about the application and its partitioning.

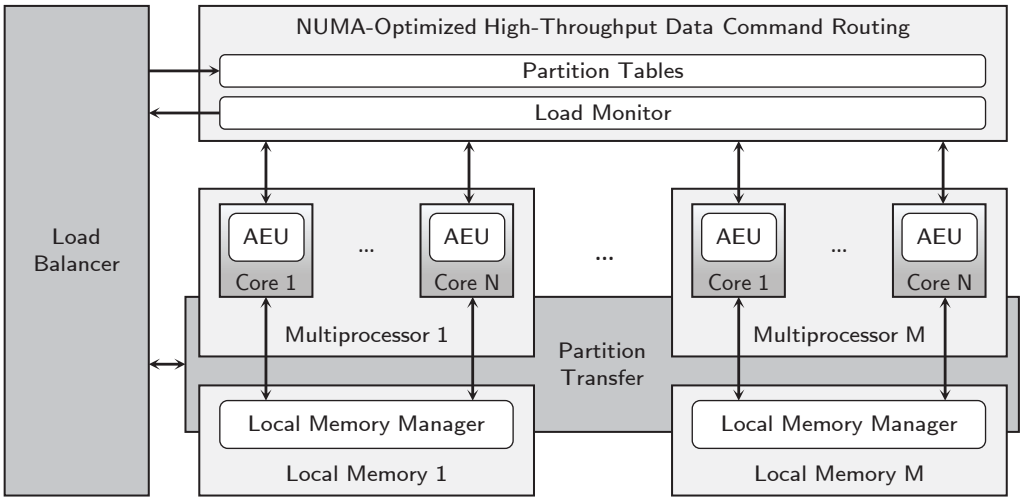


Figure 7.2.: Architectural Overview of ERIS

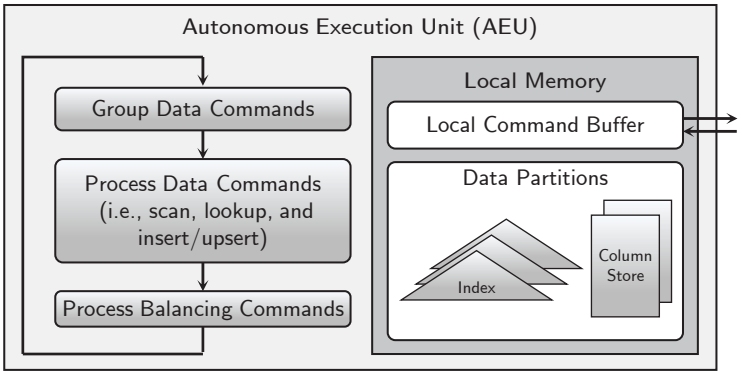


Figure 7.3.: AEU Processing and Data Organization

7.2. ERIS System Architecture

In this section, we describe the architecture of the ERIS in-memory DBMS. ERIS is optimized towards modern multiprocessor systems with non-uniform memory access. In this thesis, we describe the major component of ERIS. For a more detailed description of ERIS, e.g., of the implementation of the efficient command routing facility, we refer to Kissinger et al. (2014).

7.2.1. AEUs and Memory Management

The architecture of ERIS and individual components are visualized in Figures 7.2 and 7.3.

ERIS employs a data-oriented architecture where each data object is logically and physically partitioned. The central components of the storage engine are the worker threads, which are called Autonomous Execution Units (AEUs) in ERIS. Each available core of the system runs an AEU, which is bound to be only executed on this single core or hardware context respectively. Every single AEU gets assigned a set

of disjoint partitions—each belonging to a different data object—and is exclusively responsible for that portion of the individual data object. Figure 7.2 shows two multiprocessors with multiples cores, each running an AEU. An AEU's processing and data organization is depicted in Figure 7.3. The AEU's main task is to manage its partitions and to process incoming data commands (i.e., scans, lookups, and inserts/upserts) on these partitions. This approach restricts memory accesses of an AEU to the multiprocessor's local main memory and data objects do not have to be protected against concurrent accesses via latches.

ERIS primarily uses range partitioning to split data objects into partitions. We decided against hash partitioning because it is not order preserving and thus disallows efficient range scans and hinders efficient load-balancing techniques. In scenarios where a table is solely completely scanned, we employ physical data size partitioning instead of range partitioning, because there is no suitable attribute as partitioning criterion available. Here, ERIS only keeps track of the AEUs that actually store a partition of the corresponding data object and uses the multicast capabilities of the routing layer to distribute data commands (the routing layer is detailed in the next section).

Regarding the memory management, ERIS deploys one memory manager per multiprocessor as it is shown in Figure 7.2. Per-multiprocessor memory managers help to reduce the contention on the memory management subsystem, which is often the bottleneck during writing operations to a data object. Moreover, this approach limits allocations of AEUs to the local main memory and enables the load balancer to perform an efficient intra-node balancing (the load balancing strategy in ERIS will be revisited in a following section).

In Figure 7.3, we illustrate the AEU loop as well as the local memory organization of an AEU. The AEU keeps local data command buffers and the actual data object partitions (either stored as a column-store or an index). In the first stage of the processing loop, the AEU scans its data command buffer, which is periodically filled by the routing layer, and groups commands by the accessed data object and the command type. This optimization step is beneficial to coalesce the same type of access to the same partition. Moreover, the command grouping allows us to execute multiple index lookup or insert/upsert operations in a single batch operation to hide the main memory latency. Following the grouping step, the AEU actually processes its data command buffer, which is the most time-consuming part of the loop. Afterwards, the AEU checks its command buffer for pending balancing or transfer commands. Such commands force an AEU to grow or shrink its partition or to transfer a range of its partition to another AEU (we refer to Kissinger et al. (2014) for a detailed evaluation of the load-balancing mechanisms in ERIS).

7.2.2. High-Throughput Data Command Routing

The data command routing is an essential part of ERIS because only one AEU can answer a request for certain data and the responsible AEU has to be supplied with the according data command quickly. A data command contains, e.g., the storage operation type (i.e., scan, lookup, or insert/upsert) and necessary parameters for the storage operation (e.g., a batch of keys for the lookup or filters for a scan). The

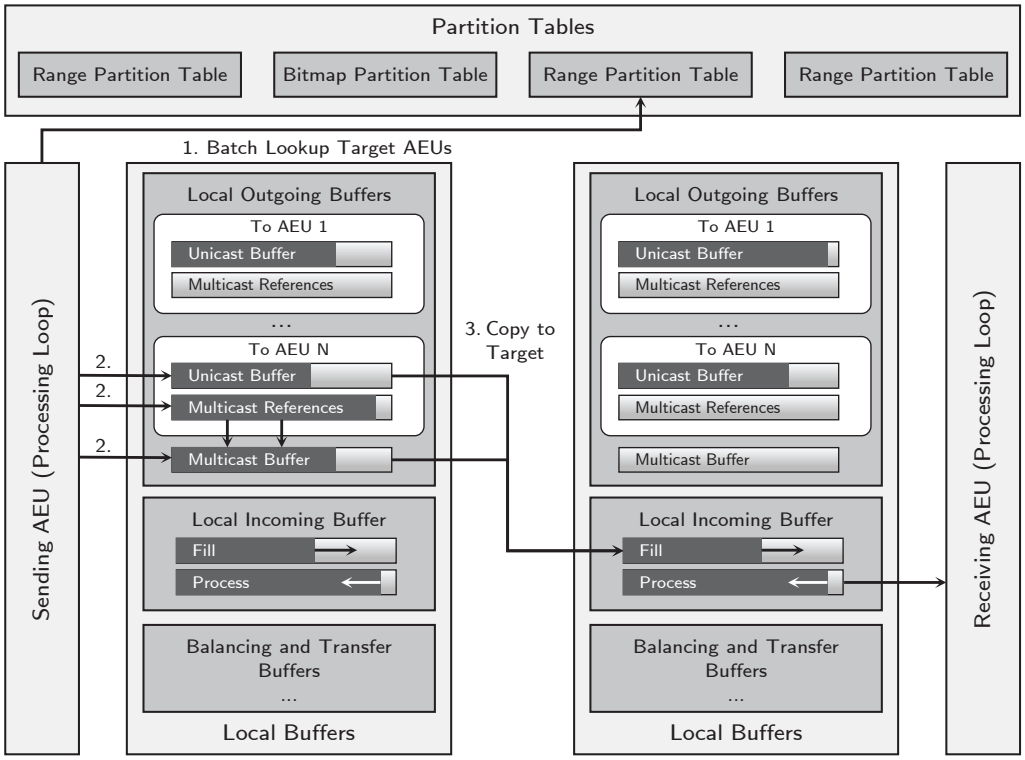


Figure 7.4.: NUMA-Optimized High-Throughput Data Command Routing

data command routing mechanism is shown in Figure 7.4. The core components are the partition tables, which keep track of the partitioning of individual data objects.

As already mentioned, a data object is either clustered on one or more of its attributes or it is distributed without any partitioning criterion. In the clustered case, the routing table stores the attribute range to AEU mapping (range partition table). If the data object is not partitioned on any attribute, the routing table only saves whether or not an AEU stores a partition of that data object (bitmap partition table). Since the routing tables are small data structure that are rarely updated (only during load balancing) and are frequently read, they usually fit in the caches of all multiprocessors and are thus not causing any remote memory accesses.

Every time an AEU generates a data command during the processing stage, it starts with a batch lookup of the responsible AEUs for that data command in the corresponding routing table of the target table (step 1 in Figure 7.4). As soon as the target AEUs are determined, the routing layer splits the command into smaller pieces, for instance, if a lookup data command contains keys that belong to different partitions. Data commands for a single AEU are written to the corresponding outgoing buffer of the source AEU (step 2 in Figure 7.4). If multiple AEUs are responsible for a data command (e.g., a scan that needs to be distributed to different AEUs), the command itself is written to the multicast buffer and references are added to the corresponding buffer. If an outgoing buffer is either full or the AEU starts over its processing loop, the specific outgoing buffer including its multicast data commands is copied to the incoming buffer of the target AEU (step 3 in Figure 7.4). This local

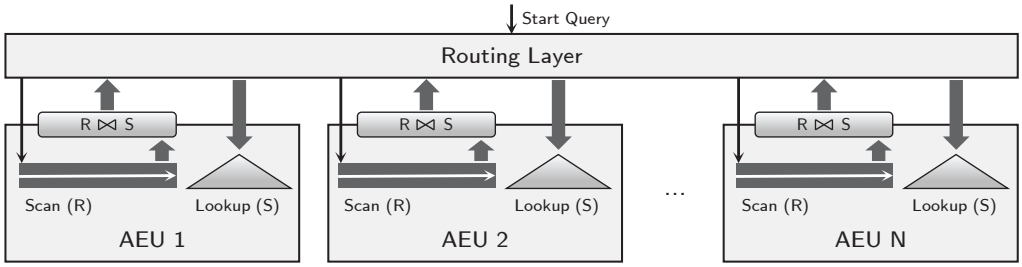


Figure 7.5.: Query Processing Model in ERIS

pre-buffering dramatically increases the data command routing throughput, because the contention on the incoming buffers is reduced and multiple data commands can be copied sequentially. Thus, the high latency of remote memory accesses on the NUMA platform does not become the bottleneck.

While outgoing buffers are private to an AEU and thus do not require any concurrency control, incoming buffers are written by different AEUs and are read by the host AEU at the same time. Hence, incoming buffers need an efficient and ideally latch-free concurrency control mechanism. We employ an adapted version of the latch-free multi-buffer proposed by Levandoski et al. (2013). For further implementation details regarding the command routing as well as an isolated performance evaluation, we refer to Kissinger et al. (2014).

7.2.3. Query Processing

In this section, we sketch the principles of NUMA-aware query processing in ERIS. Since the ERIS storage engine only supports basic storage operations (i.e., scan, lookup, and insert/upsert), a query processing model is necessary to orchestrate these storage operations and allow the execution of queries. For a more detailed description of the ERIS query processing, we again refer to Kissinger et al. (2014).

The AEUs deployed by ERIS act as data command generators as well as data command processors. Figure 7.5 visualizes an exemplary join of the tables `R` and `S`. An I/O thread has to start the query plan by issuing the scan operation. The main difference in processing a join compared to a simple scan is that an AEU does not route a result back to the operator. Instead, it invokes the callback function of the data command and is thus interleaving the execution of operator code as well as storage operations locally on the AEU. The called code directly generates new data commands, lookups in the example, which are routed to the corresponding AEUs. The newly generated data commands lookup the keys and perform the join. Furthermore, they may also directly include a callback to the successive operator, e.g., to group the result of the join. ERIS is able to execute composed operators (e.g., a 2-way-join-group-by) as proposed by the QPPT query processing model (Kissinger et al., 2013).

7.2.4. Load Balancing Strategy

Load balancing is a fundamental mechanism in ERIS given the fact that the data is partitioned and a single AEU is responsible for certain data. Skew in either the data or the accesses can cause an unbalanced system with idle resources and suboptimal performance.

If the load needs to be balanced, each participating AEU receives a balancing command. Whenever data needs to be transferred to an AEU on the same node and thus in the same memory management domain, the cheap *link* mechanism is used. The receiving AEU simply links (e.g., in case of a tree-based index) respectively appends (in case of a column-store) the new data to its own partition. In contrast, inter-node transfers use the *copy* mechanism for the partition transfer. Such a copy operation requires a cooperation between source and target AEU. The source AEU flattens the partition to an exchange format and streams it sequentially to the target AEU. The target AEU converts the data stream back to an index and links it to its existing partition. If the data object is already stored in a flat format such as a column store, the target AEU directly copies the data from the source AEU.

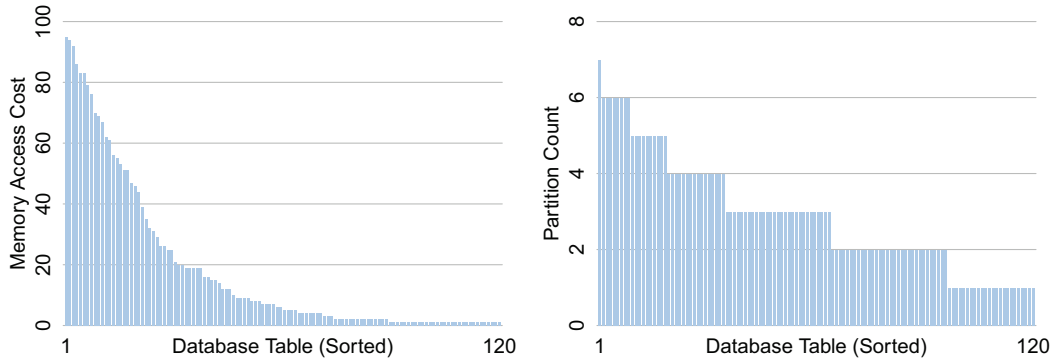
For our experiments, we implemented a two-step load balancing strategy. The allocation strategies that are investigated in this thesis see each NUMA node as a whole and assigns data objects to all AEUs that share a NUMA node. The distribution of the data objects among the AEUs of each node is up to the ERIS system and based on the observed load. Changing the data distribution inside a node uses the lightweight intra-node balancing mechanism and is constantly performed. However, changing the allocation requires the inter-node balancing mechanism and is driven by the allocation strategy.

7.3. System Performance Experiments

To evaluate the end-to-end performance of our allocation strategies, we deploy the full system, i.e., multiple parallel workloads executed in ERIS. We use two different multiprocessor systems for these experiments. We enable ERIS to deploy allocations based on different allocation strategies. Similar to the previous experiments with the MTM system, we are interested in the benefits of the penalized graph partitioning allocation strategy over a baseline and over the unmodified graph partitioning allocation strategy. The benefit will be evaluated in the overall system balance, measured in relative response times. Details about the used workloads and infrastructures as well as the performance metrics are presented in the following paragraphs. The experiment results conclude this section.

7.3.1. Workload

We again use the Star Schema Benchmark (SSB) (O'Neil et al., 2009) as a basis for our experiment workloads. Based on the database schema, we use different synthetic scan and join queries to generate typical data access and data communication patterns. The queries are evaluated using a cost model that is based on ideas from the generic database cost model for hierarchical memory systems proposed by Manegold and Boncz (2002). The authors assume perfect knowledge about the data volumes,



(a) Memory Access Cost Distribution (b) Partition Count Distribution

Figure 7.6.: Workload Characteristics

i.e. cardinalities, to which they refer as the *logical cost component*. We provide this information to our cost model by actually executing the queries in ERIS and collecting input and intermediate data statistics. Based on these statistics, the cost model quantifies the *physical cost component* of the implementations of the various operators. The physical cost depends on different characteristics of the cache/memory hierarchy, which we gathered in our low-level experiments, and the memory access patterns of the operators. Since the execution times of our benchmark queries are dominated by their memory accesses, we decided to model memory access cost as a resource in our workload model (and consequently in the infrastructure model as well).

For the experiment workload, database tables are generated in 28 different sizes ranging from 10 MiB to 1 GiB. Together with a think time between query executions, we are able to generate workload entities with any given intensity/cost. To generate a more controllable environment for our experiments, we run a static workload and each workload entity (i.e., table) executes a single benchmark query repeatedly.

To run the experiment, we generate 120 tables with random vertex weights. The vertex weights represent memory access costs and follow a Zipf distribution (exponent $s = 1$). Each table is partitioned into a random number of parts which follows a Zipf distribution between 1 and 8. The distribution of memory access costs and partition counts across the 120 tables is shown in Figure 7.6. The resulting workload graph consists of the table partitions, which are connected by edges if they belong to the same table. This graph has, depending on the actual degree of parallelism, approximately 180 vertices. Edges in the workload graph can result in distributed transactions in the actual workload, depending on the allocation. Given the vertex weights, for each vertex, a database table and benchmark query are selected that match the vertex weight best. A think time is added to match the table's memory access cost to the generated vertex weight. The table's size is added as a second weight to the vertex. Consequently, our workload model contains two resources: (1) memory access cost as an unbounded resource and (2) database size as a bounded resource.

The total size of all tables in the experiment is approximately 12 GiB.

7.3.2. Infrastructure

We use the previously introduced Intel machine and the AMD machine for the system performance experiments.[1] The same workload with 120 tables is used on both machines. The infrastructure model in our experiment is based on low-level–benchmark results. Following the workload, we model two resources in our infrastructure: (1) memory access bandwidth and (2) memory capacity. Since memory access is an unbounded resource, only relative capacities are needed in the infrastructure model, i.e., they are all equal to 1 given the identical NUMA nodes. Additionally, each node has 32 GiB (Intel machine) or 8 GiB (AMD machine) main memory which is added as the second (bounded) capacity to each node in the infrastructure graph. For the Intel machine, which is fully connected, we model a homogeneous infrastructure graph with unit capacities on all links. For the AMD machine, we model a heterogeneous infrastructure graph which is also fully connected but has link capacities according to the measured point-to-point bandwidths.[2] To minimize communication cost, we greedily map heavy edges in the partitioned workload graph to wide links in the infrastructure (see Section 5.2).

To investigate the non-linear resource usage of the infrastructure nodes, we conducted a number of synthetic scalability experiments on both machines. The ERIS system, being optimized for multiprocessor systems, scales well with the number of tasks. Consequently, only a small (linear) penalty is added to the workload cost. For the Intel machine, which has more cores per node, this penalty starts at about 20 tasks. For the AMD machine, the penalty already shows with 10 or more tasks.

7.3.3. Allocation Strategies

In this experiment, we compare the three different allocation strategies already used in the MTM experiments: (1) First Fit (FF), (2) Unmodified Graph Partitioning (UGP), and (3) Penalized Graph Partitioning (PGP).

The First Fit method (FF) acts as a simple baseline allocation strategy that (in contrast to a round-robin strategy) does not overallocate a node's memory. The First Fit method sorts all database tables by their descending size and allocates each database to the least utilized infrastructure 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) in our experiment is based on the unmodified graph partitioning algorithm in METIS. Here, we partition the workload graph (without penalty) to get a balanced allocation with minimal communication.

The third allocation strategy (PGP) uses our penalized graph partitioning algorithm in PenMETIS. The penalty function of the infrastructure model is used to describe the infrastructure behavior and to produce a better allocation.

[1] We were not able to deploy the necessary software components for the experiment on the SGI machine, which is used as a production machine in the Center for Information Services and High Performance Computing (ZIH) at the TU Dresden.

[2] Note that modeling a fully connected graph is only an approximation of the actual infrastructure. The model ignores, e.g., congestion on links that are used in multiple two-hop connections in the actual communication network.

7.3.4. Performance Metrics

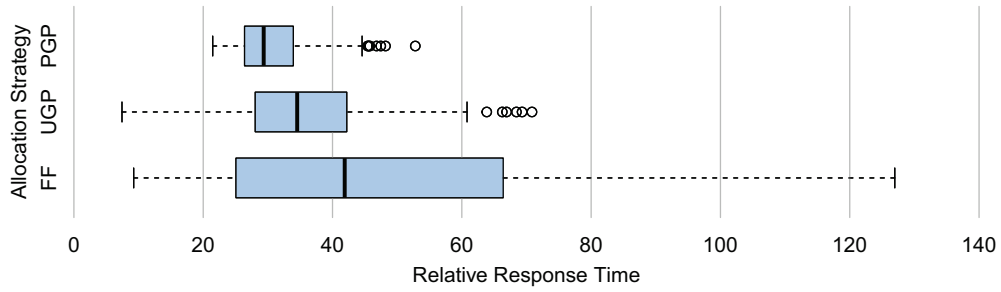
To evaluate the balance of the various setups and hence to evaluate the quality of the allocation strategies, we again use the Relative Response Time (RRT) as performance metric. We use the distribution of relative response times across all database tables to evaluate an allocation strategy. Specifically, we concentrate on three statistical measures: (1) Average Relative Response Time (AvgRRT), (2) Median Relative Response Time (MedRRT), and (3) Maximum Relative Response Time (MaxRRT). Ideally, MaxRRT should be the lowest in a balanced system while MedRRT and AvgRRT are not worse compared to other allocation strategies.

Computing the performance metrics is done in several steps. First, each absolute response time is transformed into a relative response time using a separately performed baseline run (i.e., relative to the best-case execution). Thereby, we are able to compare different transactions with different absolute response times. Second, the 95% quantile of all relative response times for a given table is computed to get a single quality measure for each database table. Using the quantile assumes that the user is interested in acceptable performance for the majority of executions. The 95% quantile is also more robust to small numbers of outliers compared to, e.g., the maximum relative response time (per database). Given one relative response time for each table, AvgRRT, MedRRT, and MaxRRT across all tables can be computed as quality measures for the whole data-oriented system.

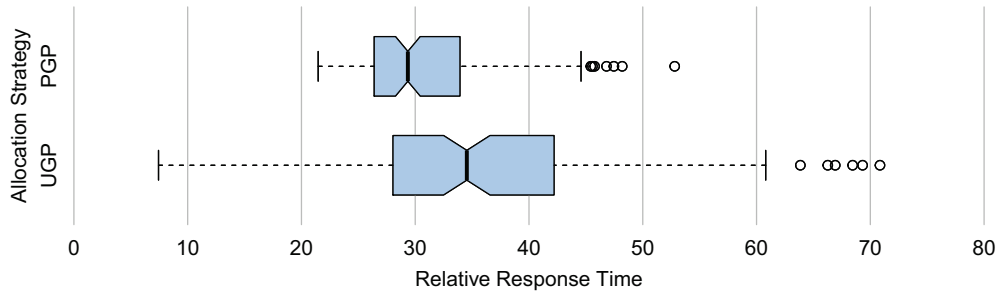
7.3.5. Experiment Results

To run the experiments, we modified ERIS to accept different allocations as parameters. For any of the three tested allocation strategies, we first load the 120 tables from raw files to (partitioned) in-memory database tables (i.e., approximately 180 partitions). Given the 4 and 8 nodes in the Intel machine and the AMD machine respectively, each node hosts on average 45 or 22.5 table partitions. Next in the workflow follows a baseline run where each benchmark query is run individually to record the best-case execution time. Finally, all benchmark queries are executed repeatedly and in parallel for 30 minutes and response times are logged for the later analysis. Using the response time log, all performance metrics are calculated.

The results of the experiments on the Intel machine are visualized in Figure 7.7 as box-whisker plots. Figure 7.7b shows the same results as Figure 7.7a on a differently scaled x-axis for a better visualization of the UGP and PGP allocation strategies. The plot in Figure 7.7a shows that the First Fit allocation strategy (FF) leads to a wide range of response times for the various benchmark queries. Consequently, while leading to a valid allocation, First Fit is unable to balance the load across the nodes. Only comparing the allocation strategies that are based on graph partitioning, it can be seen in Figure 7.7b that Penalized Graph Partitioning (PGP) leads to better overall system performance in all metrics. PGP causes fewer outliers than UGP and has better maximum, average, and median relative response times. The allocation computed by PGP is overall more balanced than the allocation computed by the UGP strategy. However, with relative response times still ranging from 21.4 to 52.8 when using the best allocation strategy, there is room for improvement. Given that the allocation is supposedly balanced, the skew in the response times is likely caused

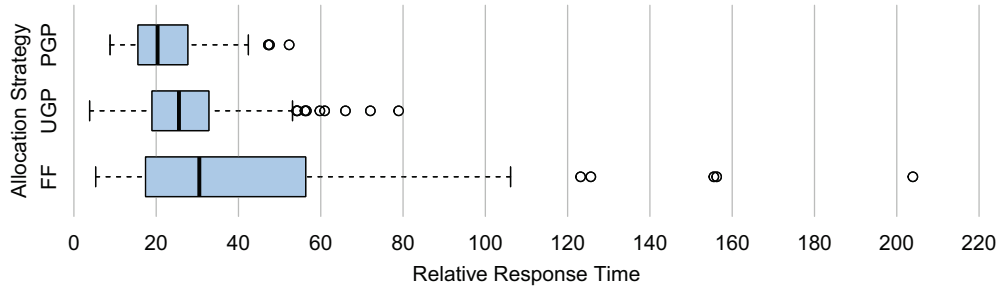


(a) Relative Response Time Distribution, All Allocation Strategies

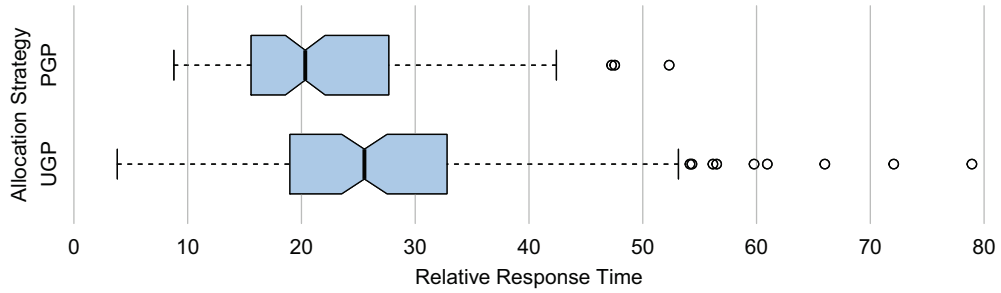


(b) Relative Response Time Distribution, Only Graph Partitioning Strategies

Figure 7.7.: Relative Response Time Distribution, Intel-Machine (Note Different X-Axes)



(a) Relative Response Time Distribution, All Allocation Strategies



(b) Relative Response Time Distribution, Only Graph Partitioning Strategies

Figure 7.8.: Relative Response Time Distribution, AMD-Machine (Note Different X-Axes)

by inaccuracies in the workload and the infrastructure model.

The results of the experiments on the AMD machine are visualized in Figure 7.8. Again, Figure 7.8b shows the same results as Figure 7.8a on a differently scaled x-axis for the UGP and PGP allocation strategies. The results on the AMD machine are in general similar to the ones on the Intel machine. The relative response times are lower compared to the Intel machine which can be explained by the higher number of nodes in the AMD machine (i.e., fewer tables share a node). The First Fit allocation strategy leads to a few particularly bad outliers and again to an overall unbalanced system. Figure 7.8b confirms that Penalized Graph Partitioning (PGP) leads to better overall system performance in all metrics compared to Unmodified Graph Partitioning (UGP).

Beyond the presented results, the experiments with the ERIS system showed that the performance is (in some cases) sensitive to small changes. Given the highly optimized code of ERIS, small skew in data or accesses quickly leads to suboptimal performance or unexpected results. Furthermore, experiments with very few tables showed that the assumption of the allocation problem, i.e., that minimizing communication yields best performance, does not always hold. With a few scan-heavy queries and sufficient capacities on the links, spreading the data out and effectively increasing the communication volume leads to the best performance. However, this behavior cannot easily be accounted for in the proposed workload and infrastructure model. It remains to future work to constantly re-evaluate the assumption on any new system and under different workloads.

7.4. Summary

We showed in this chapter, that non-uniform memory access is a defining characteristic of modern multiprocessor systems. Using low-level benchmarks on three different systems, we measured up to ten times higher latencies when accessing remote memory compared to local memory. Likewise, memory access bandwidth is up to ten times smaller on remote memory compared to local memory. Resulting from this analysis, we presented the ERIS in-memory database management system which is designed and implemented with multiprocessor systems in mind. ERIS treats the NUMA system like a distributed system and enforces fast local memory accesses. Furthermore, ERIS uses an optimized command routing mechanism to deliver data commands to the various (local) worker threads.

Our performance experiments showed that the allocation strategy, i.e., the mapping of data to NUMA nodes, strongly influences the balance of the system and consequently the worst-case performance over all tables. The allocation strategy based on penalized graph partitioning achieved the best results in our experiments. However, given the variance in the relative response times, there is still room for improvement before the system is well balanced. The allocation strategy is only as good as the workload and the infrastructure models are accurate.

Following the evaluation of our allocation strategies in the second data-oriented system in this chapter, we conclude our thesis and point out interesting directions for future work in the next chapter.

8. Conclusion

Data orientation is a common design principle in distributed data management systems. The abstract data-oriented architecture, which is based on data locality, is implemented in different systems in a variety of application scenarios. In this thesis, we presented two such scenarios, namely *database-as-a-service systems* and *DBMSs for multiprocessor systems* as well as a classification of systems therein. We later introduced the two actual data-oriented systems MTM and ERIS, one system in each of the two application scenarios. Allocation strategies are core components in any data-oriented system. Good allocation strategies can lead to balanced systems while others cause skew in the load and therefore the application performance and the infrastructure utilization. Optimal allocation strategies are hard to find given the complexity of the systems, the complicated interactions of tasks, and the huge solution space.

We showed in this thesis that data-oriented systems from different application scenarios and with different abstraction levels can be generalized to generic infrastructure and workload descriptions. We used weighted graph representations to model infrastructures with bounded and unbounded resources and possibly non-linear performance characteristics. Based on our workload and infrastructure model, we formalized the allocation problem, which seeks a valid and balanced allocation that minimizes communication costs.

Our unified allocation strategies for the generalized data-oriented system are based on the balanced k-way min-cut graph partitioning problem and the corresponding multilevel graph partitioning solution heuristic. We showed methods to solve the graph partitioning problem for single and multiple resources and proposed methods for resources with non-linear performance characteristics. On top of the basic algorithms, we proposed extensions to (1) incorporate heterogeneous infrastructures, (2) react to changing workloads and infrastructures by incrementally updating a partitioning, and (3) enable bounded resources in the infrastructure model.

Experimental evaluations of all components of our allocation strategies on synthetic workload graphs confirmed the applicability and scalability of the methods. In end-to-end–performance experiments in MTM and ERIS, we were able to prove that our allocation strategies can be used in actual data-oriented systems and that they outperform alternative allocation strategies.

To summarize our contributions we revisit the overview of related allocation strategies from Section 3.5 and compare our solution to previous approaches. As summarized in Table 8.1, our allocation strategies, which are based on the Graph Partitioning Problem (GPP), fulfill all requirements that we collected during the analysis of the allocation problem.

	Kairos	Schism	SWORD	RTP	SharedDB	GPP
Non-Linear Performance	✓	✗	✗	✓	✗	✓
Heterogeneous Infrastructures	✓	✗	✗	✓	✗	✓
Individual Resources	✗	✗	✗	✗	(✓)	✓
Communication Costs	✗	✓	✓	✗	✗	✓
Incremental Updates	✗	✗	✓	✓	✗	✓

Table 8.1.: Comparison of Allocation Strategies

Future work

In this thesis, we built a foundation for graph-partitioning based allocation strategies for generic data-oriented systems. Given the complexity of the problem, we made many assumptions in the process and sometimes only scratched complex subproblems. Therefore, we see a lot of open challenges and list a few of them that we consider most interesting to look at:

- *Non-linear Links:* Our infrastructure model is based on non-linear performance with the number of tasks. Many tasks may cause contention across the system stack, i.e., in the hardware components, the operating system, or the data-oriented system itself. However, our model and hence our allocation strategy only considers non-linearity in the nodes, not the communication network. We assume that the links of the network also congest and hence show non-linear behavior with increasing load. It will be interesting to extend the capabilities of the graph partitioning algorithms to deal with non-linearity in the links. This task is especially challenging given that edges and communication costs are treated differently compared to vertices in the graph partitioning algorithms.
- *Pro-active Balancing:* Our allocation strategies are able to react to changes in the workload or the infrastructure by incrementally updating a partitioning. Depending on the frequency of changes, the delay between the changes and the reorganization may lead to suboptimal performance. An interesting approach, though orthogonal to the allocation strategy, will be to model the workload (i.e., the tasks) as time series and to predict workload changes. The predicted changes can then pro-actively be incorporated in the allocation strategy.
- *Replication:* Replication in data-oriented systems can be used to enforce some of the possible constraints, e.g., fault-tolerance and availability. Replication of data has several aspects to consider. First, it needs to be decided which objects to replicate. Second, the number of replicas needs to be determined. Third, the execution platform needs to support replication, which includes

read and write strategies. Incorporating replication in the proposed allocation strategies will be challenging but may extend the applicability of the method to a wider set of systems and application scenarios.

- *Service Classes:* Many data-oriented systems, especially systems that implement some form of as-a-service system, have requirements to offer different service classes for different customers. Different such service classes may be incorporated in the allocation strategy, e.g., by assigning tasks to faster nodes or by artificially increasing vertex costs in the workload graph and thus requesting more resources for these tasks. Incorporating service classes in the proposed allocation strategies is a challenge for future extensions.
- *Workload and Infrastructure Model Maintenance:* The allocation strategy in a data-oriented system is only as good as the workload and the infrastructure models are accurate. Since our focus was on the allocation strategy itself, we manually built and experimentally evaluated the models in this thesis. However, improved methods to gather more precise models, possibly in an automated fashion, are challenging research topics themselves. Given the dynamics in workload and infrastructure, future work can also be based on improved methods to maintain the corresponding models in the presence of changes.

A. Additional Graph Partitioning Results

Additional results that we acquired during the experimental evaluation of PenMETIS are summarized in the tables in this Appendix.

Graph	V	E	Total Cut	Balance	Time [ms]	Mem [MB]
144	144,649	1,074,393	85,454	1.03	446	50.7
3elt	4,720	13,722	1,641	1.01	54	1.4
4elt	15,606	45,878	2,796	1.02	55	3.2
598a	110,971	741,934	61,630	1.03	339	37.5
add20	2,395	7,462	3,301	1.01	74	1.4
add32	4,960	9,462	547	1.01	47	1.4
auto	448,695	3,314,611	188,817	1.03	1,158	155.0
bcsstk29	13,992	302,748	63,518	1.02	117	16.0
bcsstk30	28,924	1,007,284	189,324	1.02	220	46.0
bcsstk31	35,588	572,914	65,126	1.02	186	27.3
bcsstk32	44,609	985,046	103,789	1.02	214	38.2
bcsstk33	8,738	291,583	111,771	1.02	211	18.5
brack2	62,631	366,559	28,804	1.03	187	20.4
crack	10,240	30,380	2,731	1.02	54	2.4
cs4	22,499	43,858	4,616	1.03	93	4.9
cti	16,840	48,232	6,552	1.03	94	4.5
data	2,851	15,093	3,316	1.01	56	1.2
fe_4elt2	11,143	32,818	2,699	1.02	53	2.5
fe_body	45,087	163,734	5,392	1.04	97	9.5
fe_ocean	143,437	409,593	24,143	1.02	260	34.2
fe_pwt	36,519	144,794	8,786	1.02	91	8.4
fe_rotor	99,617	662,431	51,467	1.03	300	34.8
fe_sphere	16,386	49,152	3,964	1.02	66	3.5
fe_tooth	78,136	452,591	38,813	1.03	201	25.4
finan512	74,752	261,120	11,056	1.03	174	20.0
m14b	214,765	1,679,018	107,395	1.03	573	80.5
memplus	17,758	54,196	17,828	1.02	137	5.8
t60k	60,005	89,440	2,448	1.02	81	10.6
uk	4,824	6,837	479	1.02	42	1.3
vibrobox	12,328	165,250	53,729	1.02	172	9.8
wave	156,317	1,059,331	93,678	1.03	437	56.9
whitaker3	9,800	28,989	2,672	1.02	55	2.3
wing_nodal	10,937	75,488	16,976	1.03	109	5.2
wing	62,032	121,544	8,864	1.03	159	12.9

Table A.1.: Partitioning Statistics METIS (64 Partitions, 3% Imbalance)

Graph	V	E	Total Cut	Balance	Time [ms]	Mem [MB]
144	144,649	1,074,393	87,901	1.03	493	58.0
3elt	4,720	13,722	1,636	1.01	70	1.6
4elt	15,606	45,878	2,744	1.02	71	3.5
598a	110,971	741,934	62,748	1.03	379	39.8
add20	2,395	7,462	3,280	1.01	93	1.5
add32	4,960	9,462	542	1.01	61	1.5
auto	448,695	3,314,611	187,926	1.03	1,409	164.7
bcsstk29	13,992	302,748	60,826	1.02	201	17.8
bcsstk30	28,924	1,007,284	187,856	1.02	239	46.6
bcsstk31	35,588	572,914	64,745	1.02	205	28.3
bcsstk32	44,609	985,046	107,893	1.02	246	40.4
bcsstk33	8,738	291,583	114,597	1.02	377	21.0
brack2	62,631	366,559	28,743	1.03	224	21.6
crack	10,240	30,380	2,765	1.02	75	2.9
cs4	22,499	43,858	4,604	1.03	114	5.5
cti	16,840	48,232	6,520	1.03	118	5.1
data	2,851	15,093	3,157	1.01	101	2.0
fe_4elt2	11,143	32,818	2,677	1.02	84	2.8
fe_body	45,087	163,734	5,585	1.02	101	10.4
fe_ocean	143,437	409,593	24,290	1.02	318	37.5
fe_pwt	36,519	144,794	8,785	1.02	111	9.2
fe_rotor	99,617	662,431	51,432	1.03	344	36.6
fe_sphere	16,386	49,152	3,981	1.02	84	3.9
fe_tooth	78,136	452,591	38,941	1.03	256	27.1
finan512	74,752	261,120	11,293	1.03	207	21.8
m14b	214,765	1,679,018	108,573	1.03	641	84.8
memplus	17,758	54,196	18,160	1.02	141	6.1
t60k	60,005	89,440	2,419	1.02	101	11.7
uk	4,824	6,837	476	1.02	61	1.4
vibrobox	12,328	165,250	52,726	1.02	251	11.3
wave	156,317	1,059,331	92,312	1.03	468	60.4
whitaker3	9,800	28,989	2,680	1.02	76	2.6
wing_nodal	10,937	75,488	16,949	1.03	155	5.3
wing	62,032	121,544	9,049	1.03	193	14.3

Table A.2.: Partitioning Statistics PenMETIS (64 Partitions, 3% Imbalance)

	Number of Partitions								
Graph	4	8	16	32	64	128	256	512	1024
144	277	300	335	377	446	599	960	1,571	2,912
3elt	4	6	11	23	54	114	173	243	311
4elt	13	14	19	30	55	115	226	463	665
598a	192	219	228	256	339	508	790	1,309	2,352
add20	6	10	20	40	74	98	125	151	171
add32	4	6	10	19	47	105	151	208	260
auto	878	932	964	1,126	1,158	1,462	2,056	2,878	4,739
bcsstk29	34	37	50	72	117	228	490	1,139	1,439
bcsstk30	93	99	117	158	220	361	729	1,587	3,631
bcsstk31	74	81	90	127	186	315	599	1,283	2,822
bcsstk32	110	120	132	158	214	319	545	1,101	2,736
bcsstk33	37	46	69	112	211	460	943	1,159	1,617
brack2	87	100	115	133	187	316	521	917	1,778
crack	11	12	19	31	54	108	238	440	621
cs4	24	29	36	56	93	169	311	550	967
cti	18	23	31	53	94	181	348	678	925
data	4	6	11	24	56	132	204	264	314
fe_4elt2	9	10	16	29	53	104	208	405	632
fe_body	41	44	48	67	97	174	299	709	1,269
fe_ocean	149	170	189	229	260	404	648	1,122	1,989
fe_pwt	32	36	41	59	91	166	293	644	1,358
fe_rotor	157	174	185	213	300	438	711	1,206	2,202
fe_sphere	14	17	24	37	66	124	250	515	703
fe_tooth	116	131	143	165	201	333	553	1,004	2,198
finan512	89	90	96	105	174	298	599	1,089	2,060
m14b	386	423	438	483	573	803	1,084	1,833	3,013
memplus	34	38	51	84	137	225	341	564	684
t60k	39	42	47	55	81	133	250	485	946
uk	3	5	10	19	42	93	137	182	233
vibrobox	38	48	73	110	172	292	564	1,205	1,638
wave	242	271	283	354	437	582	950	1,531	3,017
whitaker3	8	10	16	27	55	137	262	385	537
wing	74	80	94	111	159	254	460	799	1,411
wing_nodal	18	21	35	62	109	203	385	773	1,121

Table A.3.: Partitioning Times in Milliseconds for Various Partition Counts (METIS, 3% Imbalance)

	Number of Partitions								
Graph	4	8	16	32	64	128	256	512	1024
144	306	330	368	414	493	700	1,113	1,967	3,703
3elt	5	7	15	32	70	149	245	365	562
4elt	14	16	22	36	71	196	330	687	1,231
598a	216	232	261	307	379	564	921	1,703	4,145
add20	6	12	30	49	93	127	166	208	268
add32	6	7	12	25	61	127	196	295	405
auto	997	1,046	1,085	1,182	1,409	1,680	2,264	3,508	6,129
bcsstk29	35	39	52	79	201	463	1,051	1,547	2,235
bcsstk30	97	105	126	167	239	409	1,497	3,319	4,615
bcsstk31	78	86	95	130	205	354	1,094	1,609	3,969
bcsstk32	119	127	139	164	246	427	876	2,024	5,124
bcsstk33	39	51	71	118	377	555	1,158	1,634	2,342
brack2	98	110	121	148	224	346	625	1,348	2,540
crack	12	14	21	37	75	180	412	620	925
cs4	28	32	40	64	114	231	515	927	1,378
cti	19	25	35	61	118	297	457	946	1,793
data	4	7	18	43	101	167	251	358	477
fe_4elt2	11	12	19	35	84	180	370	602	1,036
fe_body	48	50	57	70	101	201	421	952	2,555
fe_ocean	173	190	218	256	318	509	819	1,532	2,920
fe_pwt	37	42	47	70	111	200	438	917	2,133
fe_rotor	174	194	204	252	344	489	841	1,896	3,986
fe_sphere	17	20	28	45	84	216	359	704	1,150
fe_tooth	131	146	154	188	256	411	673	1,445	2,977
finan512	99	99	108	114	207	388	954	1,602	3,151
m14b	433	468	486	562	641	860	1,286	2,170	4,177
memplus	36	41	55	92	141	307	422	687	1,021
t60k	46	50	56	67	101	175	341	854	1,654
uk	4	6	11	28	61	117	176	273	399
vibrobox	45	53	76	119	251	467	954	1,390	1,935
wave	273	302	309	384	468	661	1,091	2,025	3,920
whitaker3	9	12	19	35	76	167	345	556	968
wing	84	91	107	133	193	318	563	1,208	2,211
wing_nodal	19	23	36	68	155	324	680	1,038	1,620

Table A.4.: Partitioning Times in Milliseconds for Various Partition Counts (PenMETIS, 3% Imbalance)

Graph	Number of Constraints 1	2	3	4	5
144	392	828	901	980	1,043
3elt	46	446	564	661	761
4elt	54	431	542	632	733
598a	309	911	926	1,006	1,103
add20	62	378	486	570	669
add32	40	337	428	516	612
auto	1,167	2,034	2,211	2,497	2,515
bcsstk29	113	737	867	976	1,063
bcsstk30	215	942	1,052	1,148	1,359
bcsstk31	165	643	715	783	896
bcsstk32	201	647	688	762	850
bcsstk33	196	1,119	1,353	1,491	1,618
brack2	165	604	698	780	845
crack	55	438	553	648	749
cs4	91	518	630	735	810
cti	90	541	670	786	879
data	52	534	666	763	849
fe_4elt2	52	410	542	639	729
fe_body	98	418	502	600	675
fe_ocean	264	733	753	812	892
fe_pwt	91	418	516	601	698
fe_rotor	265	845	918	997	1,093
fe_sphere	62	436	547	649	751
fe_tooth	200	638	665	734	828
finan512	172	677	756	857	965
m14b	530	1,110	1,192	1,336	1,348
memplus	112	501	585	695	788
t60k	80	402	467	564	651
uk	37	373	476	583	679
vibrobox	161	783	875	997	1,148
wave	373	868	963	943	1,011
whitaker3	51	429	548	642	745
wing	165	600	640	713	826
wing_nodal	99	605	741	822	945

Table A.5.: Partitioning Times in Milliseconds for Various Numbers of Constraints (64 Partitions, 3% Imbalance)

Graph	Number of Constraints 1	2	3	4	5
144	86,347	87,509	88,510	88,501	89,730
3elt	1,668	1,866	2,130	2,216	2,524
4elt	2,793	3,083	3,273	3,666	4,070
598a	62,061	63,436	64,225	64,347	66,051
add20	3,448	3,507	3,845	4,239	4,377
add32	606	632	827	977	1,115
auto	187,250	190,125	188,694	187,489	188,443
bcsstk29	62,432	65,485	69,817	71,361	75,263
bcsstk30	185,948	196,643	199,997	208,878	216,551
bcsstk31	68,911	68,981	70,565	74,728	77,198
bcsstk32	102,292	110,443	117,950	127,555	128,796
bcsstk33	113,276	116,245	117,800	121,988	127,704
brack2	29,199	30,897	31,052	31,259	33,511
crack	2,761	3,065	3,194	3,528	3,877
cs4	4,636	4,763	5,003	4,936	5,174
cti	6,749	6,995	7,111	7,448	7,596
data	3,166	3,540	4,109	4,327	4,942
fe_4elt2	2,695	2,917	3,054	3,404	3,643
fe_body	5,682	6,086	6,890	7,178	7,831
fe_ocean	24,061	24,433	25,428	25,897	25,704
fe_pwt	8,790	9,245	9,509	10,257	10,497
fe_rotor	51,422	51,380	54,945	54,795	56,188
fe_sphere	3,944	4,278	4,521	4,585	4,768
fe_tooth	38,958	40,649	40,360	41,481	43,182
finan512	11,202	11,730	13,012	12,715	12,632
m14b	108,343	109,231	110,669	111,044	112,380
memplus	19,060	19,626	19,550	19,627	20,226
t60k	2,491	2,768	2,778	2,856	2,992
uk	482	546	636	748	861
vibrobox	52,587	53,972	55,859	56,237	56,477
wave	93,262	93,451	94,684	95,204	97,305
whitaker3	2,688	2,891	3,033	3,210	3,423
wing	8,923	9,294	9,461	9,665	9,750
wing_nodal	17,102	17,443	17,945	19,095	19,125

Table A.6.: Total Cut for Various Numbers of Constraints (64 Partitions, 3% Imbalance)

Graph	Imbalance Parameter 50%	10%	5%	1%	0.1%
144	389	391	426	389	424
3elt	45	45	45	44	47
4elt	54	54	54	54	59
598a	300	312	330	311	332
add20	63	62	62	65	65
add32	40	40	40	41	42
auto	1,154	1,155	1,170	1,169	1,294
bcsstk29	111	114	113	118	120
bcsstk30	201	210	209	215	218
bcsstk31	156	163	165	169	179
bcsstk32	187	196	197	202	214
bcsstk33	183	191	194	202	182
brack2	167	177	177	178	194
crack	55	55	55	56	60
cs4	91	92	91	92	93
cti	87	87	87	88	92
data	50	51	53	53	55
fe_4elt2	51	51	52	52	56
fe_body	78	78	101	102	109
fe_ocean	254	265	262	261	278
fe_pwt	88	90	90	90	111
fe_rotor	256	291	264	292	277
fe_sphere	62	63	62	61	68
fe_tooth	201	202	200	218	217
finan512	166	174	175	174	258
m14b	516	524	526	530	610
memplus	110	111	111	113	119
t60k	79	80	80	80	83
uk	38	38	38	39	39
vibrobox	148	157	155	162	148
wave	356	363	368	414	442
whitaker3	51	51	50	50	54
wing	160	159	159	162	170
wing_nodal	97	98	103	106	105

Table A.7.: Partitioning Times in Milliseconds for Various Imbalance Parameters (64 Partitions)

Graph	Imbalance Parameter 50%	10%	5%	1%	0.1%
144	82,915	84,357	86,013	86,514	90,187
3elt	1,573	1,618	1,631	1,791	2,105
4elt	2,719	2,749	2,808	2,910	3,761
598a	59,897	61,073	60,589	63,316	66,793
add20	3,261	3,356	3,412	3,624	3,870
add32	467	506	567	742	1,175
auto	183,732	185,522	188,061	193,524	194,354
bcsstk29	59,797	62,095	61,281	64,766	77,197
bcsstk30	173,279	185,558	185,220	192,356	219,318
bcsstk31	59,612	64,194	66,364	67,766	83,420
bcsstk32	99,115	105,276	104,616	109,390	121,841
bcsstk33	109,589	112,659	112,497	115,980	138,895
brack2	28,073	28,338	29,164	29,770	33,341
crack	2,592	2,709	2,682	2,923	3,808
cs4	4,517	4,612	4,579	4,715	6,123
cti	6,511	6,616	6,630	7,060	9,519
data	2,861	2,985	3,120	3,440	3,881
fe_4elt2	2,610	2,650	2,675	2,770	3,633
fe_body	4,962	5,218	5,470	6,002	7,724
fe_ocean	23,226	24,040	23,908	23,783	27,497
fe_pwt	8,613	8,651	8,697	8,987	11,415
fe_rotor	48,235	49,517	50,725	52,661	57,697
fe_sphere	3,889	3,899	3,908	4,134	5,641
fe_tooth	37,306	37,998	38,594	38,763	43,590
finan512	10,786	10,895	11,402	11,838	24,358
m14b	104,026	105,428	107,984	108,394	111,030
memplus	18,006	18,556	18,862	18,958	21,091
t60k	2,406	2,438	2,449	2,509	3,117
uk	440	462	474	544	606
vibrobox	48,607	51,475	52,289	52,813	65,020
wave	89,557	92,171	93,408	94,858	97,352
whitaker3	2,621	2,678	2,715	2,828	3,595
wing	8,751	8,834	8,882	8,950	11,805
wing_nodal	16,502	16,798	16,919	17,296	21,710

Table A.8.: Total Cut for Various Imbalance Parameters (64 Partitions)

Bibliography

Agrawal, Sanjay, Surajit Chaudhuri, and Vivek Narasayya (2000). "Automated Selection of Materialized Views and Indexes for SQL Databases". In: *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Cairo, Egypt, pp. 496–505.

Agrawal, Sanjay, Vivek Narasayya, and Beverly Yang (2004). "Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. Paris, France, pp. 359–370.

Ahmad, Mumtaz and Ivan T. Bowman (2011). "Predicting System Performance for Multi-Tenant Database Workloads". In: *Proceedings of the 4th International Workshop on Testing Database Systems (DBTest '11)*. Athens, Greece.

Amazon (2015). *Amazon Relational Database Service*. URL: `http://aws.amazon.com/rds/`.

Andreev, Konstantin and Harald Racke (2004). "Balanced Graph Partitioning". In: *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'04)*. Barcelona, Spain, pp. 120–124.

Apers, Peter M. G. (1988). "Data Allocation in Distributed Database Systems". In: *ACM Transactions on Database Systems (TODS)* 13.3, pp. 263–304.

Apers, Peter M. G., Carel A. van den Berg, Jan Flokstra, Paul W. P. J. Grefen, Martin L. Kersten, and Annita N. Wilschut (1992). "PRISMA/DB: A Parallel, Main Memory Relational DBMS". In: *Knowledge and Data Engineering* 4.6, pp. 541–554.

Aulbach, Stefan, Dean Jacobs, Alfons Kemper, and Michael Seibold (2009). "A Comparison of Flexible Schemas for Software as a service". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. Providence, Rhode Island, USA: ACM, pp. 881–888.

Aulbach, Stefan, Michael Seibold, Dean Jacobs, and Alfons Kemper (2011). "Extensibility and Data Sharing in Evolving Multi-Tenant Databases". In: *Proceedings of the 2011 IEEE International Conference on Data Engineering (ICDE '11)*. Hannover, Germany, pp. 99–110.

Bichot, Charles-Edmond and Patrick Siarry, eds. (2011). *Graph Partitioning*. Wiley.

Blagodurov, Sergey and Alexandra Fedorova (2011). "User-level Scheduling on NUMA Multicore Systems under Linux". In: *Linux Symposium*. Ottawa, Canada, pp. 81–91.

Blagodurov, Sergey, Sergey Zhuravlev, and Alexandra Fedorova (2010). "Contention-Aware Scheduling on Multicore Systems". In: *ACM Transactions on Computer Systems* 28.4.

Borkar, Shekhar and Andrew A. Chien (2011). "The Future of Microprocessors". In: *Communications of the ACM* 54.5, pp. 67–77.

Brandfass, Barbara, Thomas Alrutz, and Thomas Gerhold (2013). "Rank Reordering for MPI Communication Optimization". In: *Computers & Fluids* 80, pp. 372–380.

Buluç, Aydin, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz (2013). "Recent Advances in Graph Partitioning". In: *preprint: Computing Research Repository*, pp. 1–37. arXiv: `arXiv:1311.3144v3`.

Curino, Carlo, Evan P. C. Jones, Y. Zhang, and Sam Madden (2010). "Schism: a Workload-Driven Approach to Database Replication and Partitioning". In: *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB '10)*. Singapore, China, pp. 48–57.

Curino, Carlo, Evan P. C. Jones, Sam Madden, and Hari Balakrishnan (2011). "Workload-Aware Database Monitoring and Consolidation". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. Athens, Greece, pp. 313–324.

Dargie, Waltenegus, Anja Strunk, and Alexander Schill (2011). "Energy-Aware Service Execution". In: *Proceedings of the 36th Annual IEEE Conference on Local Computer Networks (LCN '11)*. Bonn. Germany, pp. 1064–1071.

Das, Sudipto, Vivek Narasayya, Feng Li, and Manoj Syamala (2014). "CPU Sharing Techniques for Performance Isolation in Multi-tenant Relational Database-as-a-Service". In: *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB '14)*. Hangzhou, China.

Dewitt, David J. and Jim Gray (1992). "Parallel Database Systems: The Future of High Performance Database Systems". In: *Communications of the ACM* 35.June 1992, pp. 85–98.

Diekmann, Ralf, Burkhard Monien, and Robert Preis (1995). *Using Helpful Sets to Improve Graph Bisections*. Ed. by Derbiau Frank Hsu, Arnold L. Rosenberg, and Dominique Scotteau. AMS, pp. 57–73.

Docker (2015). *Docker*. URL: `https://www.docker.com/`.

Drake, Doratha E. and Stefan Hougardy (2003). "A Simple Approximation Algorithm for the Weighted Matching Problem". In: *Information Processing Letters* 85.4, pp. 211–213.

Fettweis, Gerhard, Wolfgang E. Nagel, and Wolfgang Lehner (2012). "Pathways to Servers of the Future: Highly Adaptive Energy Efficient Computing (HAEC)". In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '12)*. Dresden, Germany, pp. 1161–1166.

Fiduccia, Charles M. and R. M. Mattheyses (1982). "A Linear-Time Heuristic for Improving Network Partitions". In: *Proceedings of the 19th Conference on Design Automation (DAC '82)*. Las Vegas, Nevada, USA, pp. 175–181.

Giannikis, Georgios, Gustavo Alonso, and Donald Kossmann (2012). "SharedDB: Killing One Thousand Queries With One Stone". In: *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB '12)*. Istanbul, Turkey, pp. 526–537.

Giceva, Jana, Gustavo Alonso, Timothy Roscoe, and Tim Harris (2014). "Deployment of Query Plans on Multicores". In: *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB '15)*. Kohala Coast, Hawaii, USA, pp. 233–244.

Hackenberg, Daniel, Daniel Molka, and Wolfgang E. Nagel (2009). "Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems". In: *Proceedigns of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '09)*. New York, New York, USA, pp. 413–422.

Hendrickson, Bruce, Robert Leland, and Rafael Van Driessche (1996). "Enhancing Data Locality by Using Terminal Propagation". In: *Proceedings of the 29th Annual Hawaii International Conference on System Sciences (HICSS '96)*. Wailea, Hawaii, USA, pp. 565–574.

Huber, Frank and Johann-Christoph Freytag (2009). "Query Processing on Multi-Core Architectures." In: *Grundlagen von Datenbanken*, pp. 27–31.

Hui, Mei, Dawei Jiang, Guoliang Li, and Yuan Zhou (2009). "Supporting Database Applications as a Service". In: *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE '09)*. Shanghai, China, pp. 832–843.

Hyafil, Laurent and Ronald L. Rivest (1973). *Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems*. Tech. rep. IRIA – Laboratoire de Recherche en Informatique et Automatique.

Hypertransport Technology Consortium (2010). *HyperTransport I/O Link Specification*.

Impala (2015). *Impala*. URL: `http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html`.

Intel (2009). *An Introduction to the Intel QuickPath Interconnect*. URL: `http://www.intel.com/technology/quickpath/introduction.pdf`.

Jacobs, Dean and Stefan Aulbach (2007). "Ruminations on multi-tenant databases". In: *Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW '07)*. Aachen, Germany, pp. 5–9.

Johnson, Ryan, Ippokratis Pandis, and Anastasia Ailamaki (2008). "Critical Sections Re-emerging Scalability Concerns for Database Storage Engines". In: *Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN '08)*. Vancouver, British Columbia, Canada, pp. 35–40.

Johnson, Ryan, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi (2009). "Shore-MT: A Scalable Storage Manager for the Multicore Era". In: *Proceedings of the 12th International Conference on Extending Database Technology (EDBT '09)*. Saint Petersburg, Russia, pp. 24–35.

Karypis, George and Vipin Kumar (1995). "Analysis of Multilevel Graph Partitioning". In: *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (SC '95)*. San Diego, California, USA.

Karypis, George and Vipin Kumar (1998a). *Multilevel Algorithms for Multi-Constraint Graph Partitioning*. Tech. rep. University of Minnesota, Department of Computer Science, pp. 1–25.

Karypis, George and Vipin Kumar (1998b). "Multilevel k-way Partitioning Scheme for Irregular Graphs". In: *Journal of Parallel and Distributed Computing* 48.1, pp. 96–129.

Kemper, Alfons and Thomas Neumann (2011). "HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots". In: *Proceedings of the 2011 IEEE International Conference on Data Engineering (ICDE '11)*. Hannover, Germany, pp. 195–206.

Kernighan, Brian Wilson and Shen Lin (1970). "An Efficient Heuristic Procedure for Partitioning Graphs". In: *Bell System Technical Journal* 49.2, pp. 291–307.

Kiefer, Tim and Wolfgang Lehner (2011). "Private Table Database Virtualization for DBaaS". In: *Proceedings of the 4th IEEE International Conference on Utility and Cloud Computing (UCC '11)*. Melbourne, Australia, pp. 328–329.

Kiefer, Tim, Benjamin Schlegel, and Wolfgang Lehner (2012). "MulTe: A Multi-Tenancy Database Benchmark Framework". In: *Proceedings of the 4th TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC '12)*. Istanbul, Turkey.

Kiefer, Tim, Benjamin Schlegel, and Wolfgang Lehner (2013). "Experimental Evaluation of NUMA Effects on Database Management Systems." In: *Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW '13)*. Magdeburg, Germany.

Kiefer, Tim, Hendrik Schön, Dirk Habich, and Wolfgang Lehner (2014). "A Query, a Minute: Evaluating Performance Isolation in Cloud Databases". In: *Proceedings of the 6th TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC '14)*. Hangzhou, China, pp. 173–187.

Kissinger, Thomas, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner (2013). "QPPT: Query Processing on Prefix Trees". In: *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR '13)*. Asilomar, California, USA.

Kissinger, Thomas, Tim Kiefer, and Benjamin Schlegel (2014). "ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workloads". In: *Proceedings of the 5th International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS '14)*. Hangzhou, China.

Kumar, K. Ashwin, Amol Deshpande, and Samir Khuller (2013). "Data Placement and Replica Selection for Improving Co-location in Distributed Environments". In: *preprint: Computing Research Repository*. arXiv: `arXiv:1302.4168v1`.

Kumar, K. Ashwin, Abdul Quamar, Amol Deshpande, and Samir Khuller (2014). "SWORD: Workload-Aware Data Placement and Replica Selection for Cloud Data Management Systems". In: *The VLDB Journal - The International Journal on Very Large Data Bases* 23.6, pp. 845–870.

Leis, Viktor, Peter Boncz, Alfons Kemper, and Thomas Neumann (2014). "Morsel-Driven Parallelism : A NUMA-Aware Query Evaluation Framework for the Many-Core Age". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. Snowbird, Utah, USA, pp. 743–754.

Levandoski, Justin, David Lomet, and Sudipta Sengupta (2013). "LLAMA: A Cache/Storage Subsystem for Modern Hardware". In: *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB '13)*. Riva del Garda, Trento, Italy.

Li, Chuanpeng, Chen Ding, and Kai Shen (2007). "Quantifying the Cost of Context Switch". In: *Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS '07)*. San Diego, California, USA.

Manegold, Stefan and Peter Boncz (2002). "Generic Database Cost Models for Hierarchical Memory Systems". In: *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*. Hong Kong, China.

Mehta, Manish and David J. Dewitt (1997). "Data Placement in Shared-Nothing Parallel Database Systems". In: *The VLDB Journal - The International Journal on Very Large Data Bases* 6.1, pp. 53–72.

Microsoft (2015). *Microsoft Windows Azure*. URL: `http://www.windowsazure.com/en-us/`.

Molka, Daniel, Dan Hackenberg, and Robert Schöne (2014). "Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer". In: *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14)*. Edinburgh, Scotland.

Moulitsas, Irene and George Karypis (2008). "Architecture Aware Partitioning Algorithms". In: *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP '08)*. Ayia Napa, Cyprus, pp. 42–53.

Narasayya, Vivek, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri (2013). "SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service". In: *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR '13)*. Asilomar, California, USA.

Nehme, Rimma and Nicolas Bruno (2011). "Automated Partitioning Design in Parallel Database Systems". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. Athens, Greece, pp. 1137–1148.

O'Neil, Patrick, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak (2009). "The Star Schema Benchmark and Augmented Fact Table Indexing". In: *Proceedings of the 1st TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC '09)*. Lyon, France.

Oracle (2015). *Oracle Database Cloud Service*. URL: `https://cloud.oracle.com/database?tabID=1383678914614`.

Pandis, Ippokratis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki (2010). "Data-Oriented Transaction Execution". In: *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB '10)*. Singapore, China.

Pandis, Ippokratis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki (2011). "PLP: Page Latch-free Shared-everything OLTP". In: *Proceedings of the 37th International Conference on Very Large Data Bases (VLDB '11)*. Seattle, Washington, USA.

Pavlo, Andrew, Carlo Curino, and Stan Zdonik (2012). "Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. Scottsdale, Arizona, USA, pp. 61–72.

Pellegrini, Francois (1994). "Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs". In: *Proceedings of the IEEE Scalable High Performance Computing Conference (SHPCC '94)*. Knoxville, Tennessee, USA, pp. 486–493.

Porobic, Danica, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki (2012). "OLTP on Hardware Islands". In: *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB '12)*. Istanbul, Turkey, pp. 1447–1458.

Porobic, Danica, Erietta Liarou, Pinar Tözün, and Anastasia Ailamaki (2014). "ATraPos: Adaptive Transaction Processing on Hardware Islands". In: *Proceedings of the 2014 IEEE International Conference on Data Engineering (ICDE '14)*. Chicago, Illinois, USA, pp. 688–699.

Quamar, Abdul, K. Ashwin Kumar, and Amol Deshpande (2013). "SWORD: Scalable Workload-Aware Data Placement for Transactional Workloads". In: *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*. Genoa, Italy, pp. 430–441.

Rahm, Erhard and Robert Marek (1993). "Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems." In: *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB '93)*. Dublin, Ireland, pp. 182–193.

Rao, Jun, Chun Zhang, Nimrod Megiddo, and Guy Lohman (2002). "Automating Physical Database Design in a Parallel Database". In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. Madison, Wisconsin, USA, pp. 558–569.

Ries, D. and R. Epstein (1978). *Evaluation of Distribution Criteria for Distributed Database Systems*. Tech. rep. UC Berkeley.

Rybina, Kateryna and Waltenegus Dargie (2013). "Investigation into the Energy Cost of Live Migration of Virtual Machines". In: *Proceedings of the Third IFIP Conference on Sustainable Internet and ICT for Sustainability (SUSTAINIT '13)*. Palermo, Italy.

Rybina, Kateryna, Abhinandan Patni, and Alexander Schill (2014). "Analysing the Migration Time of Live Migration of Multiple Virtual Machines". In: *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER '14)*. Barcelona, Spain, pp. 590–597.

SGI (2012). *Technical Advances in the SGI UV Architecture*. Tech. rep.

Salomie, Tudor-Ioan, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso (2011). "Database Engines on Multicores, Why Parallelize When You Can Distribute?" In: *Proceedings of the 6th European Conference on Computer Systems (EuroSys '11)*. Salzburg, Austria, pp. 17–30.

Sanders, Peter and Christian Schulz (2013). "Think Locally, Act Globally: Highly Balanced Graph Partitioning". In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA '13)*. Rome, Italy, pp. 164–175.

Schaffner, Jan, Benjamin Eckart, Dean Jacobs, Hasso Plattner, and Alexander Zeier (2011). "Predicting In-Memory Database Performance for Automating Cluster Management Tasks". In: *Proceedings of the 2011 IEEE International Conference on Data Engineering (ICDE '11)*. Hannover, Germany, pp. 1264–1275.

Schaffner, Jan, Tim Januschowski, Megan Kercher, Tim Kraska, Hasso Plattner, Michael J. Franklin, and Dean Jacobs (2013). "RTP: Robust Tenant Placement for Elastic In-Memory Database Clusters". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. New York, New York, USA, pp. 773–784.

Scheuermann, Peter, Gerhard Weikum, and Peter Zabback (1998). "Data Partitioning and Load Balancing in Parallel Disk Systems". In: *The VLDB Journal - The International Journal on Very Large Data Bases* 7.1, pp. 48–66.

Schloegel, Kirk, George Karypis, and Vipin Kumar (1997). "Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes". In: *Journal of Parallel and Distributed Computing* 47.2, pp. 109–124.

Schloegel, Kirk, George Karypis, and Vipin Kumar (2000). "A Unified Algorithm for Load-Balancing Adaptive Scientific Simulations". In: *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (SC '00)*. Dallas, Texas, USA.

Sigoure, Benoit (2010). *How long does it take to make a context switch?* URL: `http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html`.

Soper, Alan J., Chris Walshaw, and Mark Cross (2004). "A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning". In: *Journal of Global Optimization* 29.2, pp. 225–241.

Stonebraker, Michael, Sam Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland (2007). "The End of an Architectural Era (It's Time for a Complete Rewrite)". In: *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. Vienna, Austria, pp. 1150–1160.

Strunk, Anja (2012). "Costs of Virtual Machine Live Migration: A Survey". In: *Proceedings of the 2012 IEEE Eighth World Congress on Services (SERVICES '12)*. Honululu, Hawaii, USA, pp. 323–329.

TPoX (2015). *Transaction Processing over XML (TPoX)*. URL: `http://tpox.sourceforge.net/`.

Tözün, Pinar, Brian Gold, and Anastasia Ailamaki (2013). "OLTP in Wonderland: Where Do Cache Misses Come From in Major OLTP Components?" In: *Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN '13)*. New York, New York, USA.

Walshaw, Chris (2010). "Variable Partition Inertia: Graph Repartitioning and Load Balancing for Adaptive Meshes". In: *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*. Ed. by Manish Parashar and Xiaolin Li, pp. 357–380.

Walshaw, Chris and Mark Cross (2001). "Multilevel Mesh Partitioning for Heterogeneous Communication Networks". In: *Future Generation Computer Systems* 17.5, pp. 601–623.

Wang, Zhi Hu, Chang Jie Guo, Bo Gao, Wei Sun, Zhen Zhang, and Wen Hao An (2008). "A Study and Performance Evaluation of the Multi-Tenant Data Tier Design Patterns for Service Oriented Computing". In: *Proceedings of the 2008 IEEE International Conference on e-Business Engineering (ICEBE '08)*. Xi'an, China, pp. 94–101.

Zhou, Jingren, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib (2012). "SCOPE: Parallel Databases Meet MapReduce". In: *The VLDB Journal - The International Journal on Very Large Data Bases* 21.5, pp. 611–636.

Zilio, Daniel C. (1997). "Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems". PhD thesis. University of Toronto.

Zilio, Daniel C., Anant Jhingran, and Sriram Padmanabhan (1994). *Partitioning Key Selection for a Shared-Nothing Parallel Database System*. Tech. rep. IBM Research, pp. 1–27.

Zilio, Daniel C., Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden (2004). "DB2 Design Advisor: Integrated Automatic Physical Database Design". In: *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)*. Toronto, Canada, pp. 1087–1097.

List of Figures

List of Tables