Lehrstuhl für Verteilte Informationssysteme
Fakultät für Informatik und Mathematik
Universität Passau

**Doctoral Thesis**

# Efficiency in Cluster Database Systems

## Dynamic and Workload-Aware Scaling and Allocation

Dipl. Inf. Tilmann Rabl

July 21, 2011

Advisor: Prof. Dr. Harald Kosch
Second Advisor: Prof. Lionel Brunie

*For Michael Hendrik Rabl*

# Abstract

Database systems have been vital in all forms of data processing for a long time. In recent years, the amount of processed data has been growing dramatically, even in small projects. Nevertheless, database management systems tend to be static in terms of size and performance which makes scaling a difficult and expensive task. Because of performance and especially cost advantages more and more installed systems have a shared nothing cluster architecture. Due to the massive parallelism of the hardware programming paradigms from high performance computing are translated into data processing. Database research struggles to keep up with this trend. A key feature of traditional database systems is to provide transparent access to the stored data. This introduces data dependencies and increases system complexity and inter process communication. Therefore, many developers are exchanging this feature for a better scalability. However, explicitly managing the data distribution and data flow requires a deep understanding of the distributed system and reduces the possibilities for automatic and autonomic optimization. In this thesis we present an approach for database system scaling and allocation that features good scalability although it keeps the data distribution transparent.

The first part of this thesis analyzes the challenges and opportunities for self-scaling database management systems in cluster environments. Scalability is a major concern of Internet based applications. Access peaks that overload the application are a financial risk. Therefore, systems are usually configured to be able to process peaks at any given moment. As a result, server systems often have a very low utilization. In distributed systems the efficiency can be increased by adapting the number of nodes to the current workload. We propose a processing model and an architecture that allows efficient self-scaling of cluster database systems. In the second part we consider different allocation approaches. To increase the efficiency we present a workload-aware, query-centric model. The approach is formalized; optimal and heuristic algorithms are presented. The algorithms optimize the data distribution for local query execution and balance the workload according to the query history. We present different query classification schemes for different forms of partitioning. The approach is evaluated for OLTP and OLAP style workloads. It is shown that variants of the approach scale well for both fields of application. The third part of the thesis considers benchmarks for large, adaptive systems. First, we present a data generator for cloud-sized applications. Due to its architecture the data generator can easily be extended and configured. A key feature is the high degree of parallelism that makes linear speedup for arbitrary numbers of nodes possible. To simulate systems with user interaction, we have analyzed a productive online e-learning management system. Based on our findings, we present a model for workload generation that considers the temporal dependency of user interaction.

# Kurzzusammenfassung

Datenbanksysteme sind seit langem die Grundlage für alle Arten von Informationsverarbeitung. In den letzten Jahren ist das Datenaufkommen selbst in kleinen Projekten dramatisch angestiegen. Dennoch sind viele Datenbanksysteme statisch in Bezug auf ihre Kapazität und Verarbeitungsgeschwindigkeit was die Skalierung aufwendig und teuer macht. Aufgrund der guten Geschwindigkeit und vor allem aus Kostengründen haben immer mehr Systeme eine *Shared-Nothing*-Architektur, bestehen also aus unabhängigen, lose gekoppelten Rechnerknoten. Da dieses Konstruktionsprinzip einen sehr hohen Grad an Parallelität aufweist, werden zunehmend Programmierparadigmen aus dem klassischen Hochleistungsrechen für die Informationsverarbeitung eingesetzt. Dieser Trend stellt die Datenbankforschung vor große Herausforderungen. Eine der grundlegenden Eigenschaften traditioneller Datenbanksysteme ist der transparente Zugriff zu den gespeicherten Daten, der es dem Nutzer erlaubt unabhängig von der internen Organisation auf die Daten zuzugreifen. Die resultierende Unabhängigkeit führt zu Abhängigkeiten in den Daten und erhöht die Komplexität der Systeme und der Kommunikation zwischen einzelnen Prozessen. Daher wird Transparenz von vielen Entwicklern für eine bessere Skalierbarkeit geopfert. Diese Entscheidung führt dazu, dass der die Datenorganisation und der Datenfluss explizit behandelt werden muss, was die Möglichkeiten für eine automatische und autonome Optimierung des Systems einschränkt. Der in dieser Arbeit vorgestellte Ansatz zur Skalierung und Allokation erhält den transparenten Zugriff und zeichnet sich dabei durch seine vollständige Automatisierbarkeit und sehr gute Skalierbarkeit aus.

Im ersten Teil dieser Dissertation werden die Herausforderungen und Chancen für selbst-skalierende Datenbankmanagementsysteme behandelt, die in auf Computerclustern betrieben werden. Gute Skalierbarkeit ist eine notwendige Eigenschaft für Anwendungen, die über das Internet zugreifbar sind. Lastspitzen im Zugriff, die die Anwendung überladen stellen ein finanzielles Risiko dar. Deshalb werden Systeme so konfiguriert, dass sie eventuelle Lastspitzen zu jedem Zeitpunkt verarbeiten können. Das führt meist zu einer im Schnitt sehr geringen Auslastung der unterliegenden Systeme. Eine Möglichkeit dieser Ineffizienz entgegen zu steuern ist es die Anzahl der verwendeten Rechnerknoten an die vorliegende Last anzupassen. In dieser Dissertation werden ein Modell und eine Architektur für die Anfrageverarbeitung vorgestellt, mit denen es möglich ist Datenbanksysteme auf Clusterrechnern einfach und effizient zu skalieren. Im zweiten Teil der Arbeit werden verschieden Möglichkeiten für die Datenverteilung behandelt. Um die Effizienz zu steigern wird ein Modell verwendet, das die Lastverteilung im Anfragestrom berücksichtigt. Der Ansatz ist formalisiert und optimale und heuristische Lösungen werden präsentiert. Die vorgestellten Algorithmen optimieren die Datenverteilung für eine lokale Ausführung aller Anfragen und balancieren die Last auf den Rechnerknoten. Es

werden unterschiedliche Arten der Anfrageklassifizierung vorgestellt, die zu verschiedenen Arten von Partitionierung führen. Der Ansatz wird für sowohl für Onlinetransaktionsverarbeitung, als auch Onlinedatenanalyse evaluiert. Die Evaluierung zeigt, dass der Ansatz für beide Felder sehr gut skaliert. Im letzten Teil der Arbeit werden verschiedene Techniken für die Leistungsmessung von großen, adaptiven Systemen präsentiert. Zunächst wird ein Datengenerierungsansatz gezeigt, der es ermöglicht sehr große Datenmengen völlig parallel zu erzeugen. Um die Benutzerinteraktion von Onlinesystemen zu simulieren wurde ein produktives E-learningsystem analysiert. Anhand der Analyse wurde ein Modell für die Generierung von Arbeitslasten erstellt, das die zeitlichen Abhängigkeiten von Benutzerinteraktion berücksichtigt.

# Acknowledgements

This thesis would not have been possible without the encouragement and supervision of Harald Kosch. I am grateful for his constant support and friendly advise. He allowed me the room to work in my own way and kept me motivated throughout the thesis. Furthermore, he gave me the opportunity to work at his chair which was a great experience.

In the five years at the chair I had the pleasure to work with many friendly people, who I count among my friends. I would like to thank, Günther Hölbling, who shared a room with me and with whom I had plenty of fruitful discussions. Mario Döller always impressed me and encouraged me with his effectiveness and efficiency. Florian Stegmaier brightened my day and always lent me his ear. Stella Stars, Britta Meixner, and David Coquil enriched my day with stimulating conversation. Thanks to all the members of the doctoral college, Christian, Getnet, Hatem, Lyes, Natacha, Tobias, Vanessa, and Zeina. I would especially like to thank Ingrid Winter, who was like a mother and always kept me free of nasty paper work.

I had the chance to advise many students, who helped me with my projects and motivated me with their effort. I am thankful to all of them, it was a pleasure to work with each one. I would like to thank Christoph Koch and Marc Pfeffer, who were my first students and set the bar high for the following. Marc built a first prototype for my thesis project. Bastian Hösch built a second prototype and helped me with the linear program. Marco Sitzberger examined the periodic behavior of the Stud.IP logs. Andreas Brandl helped me with the implementation of the final prototype. Christian Dellwo implemented the Scalileo framework and Niklas Schmidtmer integrated it in my prototype. Michael Frank implemented PDGF and Manuel Danisch currently adapts it for the TPC-DI benchmark. Andreas Lang helped me with the analysis of the Stud.IP logs for the workload generation. I had many more students, who did a great job on their theses.

I would also like to thank Lionel Brunie for being my supervisor. During the work on this thesis I had the opportunity to work with many nice people. Bernhard Sick worked with me on the workload generation and helped me with the mathematical background. Meikel Pöss brought me in contact with the TPC and kept me motivated to work on the benchmarking research.

Last but not least, I would like to thank my family. I am happy to live among so many fine people. Jane Mortimer spent here valuable free time and ironed out my problems with the English language. I am grateful for the constant support of my mother. Most importantly, I wish to thank the love of my life Maria and my wonderful son Maximilian. They encouraged me, supported me and gave me strength when times were rough.

I dedicate this thesis to my father. Without his support I could never have finished it.
I wish he had lived to see the end of it.

# Contents


**I. Introduction** **1**

**1. Introduction** **3**
1.1. Motivation . . . 4
1.2. Contributions . . . 5
1.3. Overview . . . 5

**2. Preliminaries** **7**
2.1. Set . . . 7
2.2. Function . . . 7
2.3. Family of Sets . . . 7
2.4. Multiset . . . 8
2.5. Sequence . . . 8
2.6. Graph . . . 8

**3. Database, Database System and Database Management System** **9**
3.1. Relational Model . . . 10
3.1.1. Relational Algebra . . . 12
3.1.2. SQL . . . 14
3.2. Architecture of a Database Management System . . . 15
3.3. Distributed Database Systems . . . 18
3.4. CDBS Processing Model . . . 21
3.4.1. Limitations of the Model . . . 24
3.4.2. Transactions . . . 25
3.5. Scientific and Commercial CDBSs . . . 26
3.5.1. C-JDBC . . . 26
3.5.2. Ganymed . . . 28
3.5.3. MIDDLE-R . . . 28
3.5.4. MySQL Cluster . . . 29
3.5.5. NonStop SQL . . . 30
3.5.6. DB2 . . . 30
3.5.7. Discussion . . . 31

**II. Scaling** 33

**4. Scaling Distributed Database Systems** 35
4.1. Automatic CDBS Scaling . . . 37
4.2. Efficiency of Distributed Systems . . . 39
4.3. Energy Efficiency of Scaling . . . 43
4.4. Autonomic Computing . . . 44

**5. Scalileo** 47
5.1. Scalileo's Architecture . . . 47
5.1.1. Workers . . . 47
5.1.2. Master . . . 48
5.1.3. Parameterized Components . . . 51
5.1.4. Benchmarks . . . 52
5.1.5. Reduction . . . 54
5.1.6. Conditions . . . 54
5.1.7. Constraints . . . 55
5.1.8. Login Methods . . . 56
5.2. Web Server Application . . . 57
5.3. Other Scaling Frameworks . . . 61
5.4. Frameworks for Energy Efficiency . . . 61
5.5. Conclusion . . . 62

**6. Autonomic Scaling for CDBSs** 63
6.1. Sensors . . . 63
6.2. Knowledge . . . 64
6.3. Effectors . . . 66
6.4. Evaluation . . . 67
6.5. Research Projects in Autonomic Scaling . . . 72
6.5.1. Ganymed . . . 72
6.5.2. KNN Prediction . . . 72
6.5.3. Sprint . . . 72
6.5.4. WattDB . . . 73
6.6. Discussion . . . 74

**III. Allocation** 75

**7. Distributed Database Layout** 77
7.1. Partitioning . . . 78
7.1.1. Vertical Partitioning . . . 79
7.1.2. Horizontal Partitioning . . . 80
7.1.3. Hybrid Partitioning . . . 84

7.2. Replication . . . 85
7.3. Allocation . . . 86

**8. Related Work** **87**
8.1. Partitioning . . . 88
8.2. Allocation . . . 90
8.3. Integrated Allocation Strategies . . . 91
8.4. Discussion . . . 92

**9. Automatic Allocation** **93**
9.1. Autonomic Allocation . . . 98
9.2. Discussion . . . 99

**10. Query Classification** **101**
10.1. Formal Definition . . . 102
10.2. Relation Based Classification . . . 104
10.3. Attribute Based Classification . . . 104
10.4. Predicate Based Classification . . . 105
10.5. Hybrid Classification . . . 105
10.6. Discussion . . . 106

**11. Allocation – Read Mostly** **107**
11.1. Formal Definition . . . 107
11.2. Optimal Allocation . . . 109
11.3. NP-Hardness of the Allocation . . . 112
11.4. Greedy Heuristic . . . 113
11.5. Meta Heuristics . . . 118
11.5.1. Evolutionary Algorithm . . . 119
11.5.2. Mutation . . . 120
11.5.3. Local Improvement . . . 120
11.6. Discussion . . . 122

**12. Considering Updates** **123**
12.1. Formal Definition - Update Considering . . . 123
12.2. Maximum Speedup . . . 125
12.3. Proof of NP-Hardness . . . 128
12.4. Optimal Allocation . . . 128
12.5. Greedy Heuristic . . . 133
12.5.1. Mutation . . . 139
12.5.2. Local Improvement . . . 140
12.6. Discussion . . . 142

**13. K-Safety** **143**
13.1. Redundant Fragments . . . 143

13.2. Redundant Query Classes . . . 144
13.3. Discussion . . . 146

**14. Physical Allocation** **149**
14.1. Implementing Scaling . . . 152
14.2. Discussion . . . 153

**15. Evaluation** **155**
15.1. TPC-H . . . 157
15.2. TPC-App . . . 159
15.3. Discussion . . . 162

**16. Summary** **165**

**IV. Benchmarking** **167**

**17. Benchmarks** **169**
17.1. TPC Benchmark™H . . . 170
17.2. TPC Benchmark™App . . . 171
17.3. E-Learning Benchmark . . . 173

**18. Benchmarking Large Dynamic Systems** **179**
18.1. Data Generation . . . 181
18.1.1. Parallel Random Number Generation . . . 182
18.1.2. Deterministic Data Generation . . . 183
18.1.3. Implementation . . . 185
18.1.4. Performance . . . 189
18.2. Workload Generation . . . 191
18.2.1. Scaling Time . . . 194
18.3. Benchmarking Objectives . . . 195
18.3.1. Basic Performance . . . 195
18.3.2. Adaptability . . . 195
18.3.3. Robustness . . . 196
18.4. Discussion . . . 196

**V. Conclusion** **199**

**19. Conclusion** **201**

**20. Ongoing and Future Work** **203**
20.1. Scaling . . . 203
20.2. Allocation . . . 203

20.3. Benchmarking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 204

# List of Figures


3.1. Database System Overview . . . . . . . . . . 9
3.2. ANSI/SPARC Reference Model . . . . . . . . . . 10
3.3. Algebraic Operator Tree . . . . . . . . . . 14
3.4. The 5 Layer DBMS Architecture Model . . . . . . . . . . 16
3.5. Logical Operator Tree . . . . . . . . . . 17
3.6. Physical Operator Tree . . . . . . . . . . 17
3.7. Client Server Architecture . . . . . . . . . . 19
3.8. Cluster Database System Architecture . . . . . . . . . . 21
3.9. Intra-Query Parallelism (left) vs. Inter-Query Parallelism (right) . . . . . 22
3.10. Architecture of a C-JDBC Cluster with RAIDb Level 2 . . . . . . . . . . 27
3.11. Overview of the Ganymed Architecture . . . . . . . . . . 28
3.12. MIDDLE-R Architecture . . . . . . . . . . 29
3.13. Architecture of the MySQL Cluster . . . . . . . . . . 30

4.1. Vertical (above) vs. Horizontal (below) Scaling . . . . . . . . . . 37
4.2. Requests per Second at the Wikimedia clusters in October 2009 in Europe (green) and the USA (blue) (image source: `http://en.wikipedia.org/wiki/Most_viewed_article`). . . . . . . . . . 43
4.3. The MAPE Loop . . . . . . . . . . 45

5.1. Overview of the Scalileo Architecture . . . . . . . . . . 48
5.2. Scalileo's Feedback Control Loop . . . . . . . . . . 53
5.3. Scalileo Web Application Setup . . . . . . . . . . 58
5.4. HTTP Workload Trace of Stud.IP at University of Passau for the First Day of the Winter Term 2009 . . . . . . . . . . 59
5.5. Energy Consumption Compared to Workload . . . . . . . . . . 60
5.6. Number of Active Servers Compared to Workload . . . . . . . . . . 60

6.1. Decision Tree for Scaling Up on High System Utilization . . . . . . . . . . 65
6.2. Scaling Procedure . . . . . . . . . . 67
6.3. Architecture of the Autonomic Scaling CDBS . . . . . . . . . . 68
6.4. Trace of Dynamic HTTP Accesses of Stud.IP at University of Passau for the Second Day of the Winter Term 2009 . . . . . . . . . . 69
6.5. Energy Consumption Compared to Workload . . . . . . . . . . 70
6.6. Number of Active Servers Compared to Workload . . . . . . . . . . 70
6.7. Average Response Time Compared to Workload . . . . . . . . . . 71

6.8. Sprint Architecture . . . . . . . . . . 73

7.1. Adaption of the ANSI/SPARC Reference Model for Distributed Systems . 77
7.2. Schematic of Vertical Partitioning . . . . . . . . . . 79
7.3. Schematic of Horizontal Partitioning . . . . . . . . . . 79
7.4. Partitioning and Allocation . . . . . . . . . . 86

9.1. Allocation of Read-Only Query Classes on 1 to 4 Nodes . . . . . . . . . . 95
9.2. Allocation of Read and Update Query Classes on 1 to 4 Nodes . . . . . . 97
9.3. Allocation in the MAPE Model . . . . . . . . . . 98

11.1. Allocation of Read-Only Query Classes on Heterogeneous Backends . . . . 109
11.2. Example for the 3-Partition Problem. . . . . . . . . . . 112
11.3. Cluster Allocation Solution to the 3-Partition Problem . . . . . . . . . . 113
11.4. Allocation of Different-Sized Tables on Heterogeneous Backends with (below) and without (above) Consideration of the Size . . . . . . . . . . 115
11.5. Heuristic (above) vs. Optimal (below) Allocation . . . . . . . . . . 118

12.1. Optimal Update Aware Allocations on Homogeneous Backends (above) and Heterogeneous Backends (below) . . . . . . . . . . 126
12.2. Example of the Bin-Packing Problem . . . . . . . . . . 128
12.3. Heuristic (above) vs Optimal (below) Allocation . . . . . . . . . . 139

14.1. Complete Bipartite Graph of the New Allocation (above) and the Existing Configuration (below) . . . . . . . . . . 150
14.2. Optimal Matching of the New Allocation (above) and the Existing Configuration (below) . . . . . . . . . . 150
14.3. Optimal Matching for a New Allocation (above) and an Existing Configuration (below) . . . . . . . . . . 151
14.4. Complete Bipartite Graph for the Mapping between a Scaled Allocation (above) and a New Hardware Configuration (below) . . . . . . . . . . 153
14.5. Optimal Mapping between a Scaled Allocation (above) and a New Hardware Configuration (below) . . . . . . . . . . 153

15.1. Architecture of the First Prototype . . . . . . . . . . 156
15.2. Architecture of the Second Prototype . . . . . . . . . . 157
15.3. TPC-H Throughput for Different Cluster Sizes . . . . . . . . . . 158
15.4. Deviation of the Throughput of the Column Based Allocation . . . . . . . 158
15.5. Degree of Replication for Different Cluster Sizes . . . . . . . . . . 159
15.6. Duration of the Allocation for Different Cluster Sizes . . . . . . . . . . 160
15.7. TPC-APP Speedup for Different Cluster Sizes . . . . . . . . . . 161
15.8. TPC-App Throughput for Different Cluster Sizes . . . . . . . . . . 162
15.9. Deviation of the Column Based Allocation . . . . . . . . . . 162

17.1. Schema of the TPC Benchmark™H . . . . . . . . . . . . . . . . . . . . . 170
17.2. Schema of the TPC Benchmark™App . . . . . . . . . . . . . . . . . . . . 172
17.3. Excerpt of the E-Learning Benchmark Schema . . . . . . . . . . . . . . . 174
17.4. The Reference Distribution in the Table Seminar_User . . . . . . . . . . . 175
17.5. Distribution of Seminars per User in Table Seminar_User . . . . . . . . . . 176
17.6. Distribution of Users per Seminar in Table Seminar_User . . . . . . . . . . 176
17.7. Most Accessed Websites in June 2008 per 6 Hours . . . . . . . . . . . . . 176

18.1. Most Accessed Websites in June 2008, Average Day per 10 Minutes . . . 180
18.2. Hierarchical Seeding Strategy . . . . . . . . . . . . . . . . . . . . . . . . 184
18.3. Hierarchical Seeding Strategy for References . . . . . . . . . . . . . . . . . 184
18.4. Architecture of the Parallel Data Generation Framework . . . . . . . . . . 186
18.5. Scaleup Results for 1 to 16 Nodes for a 100 GB SetQuery Data Set . . . . 189
18.6. Generation Time and Speed for Different Scaling Factors of the SetQuery Data Set . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190
18.7. Comparison of the Generation Speed of dbgen and PDGF . . . . . . . . . 191
18.8. Generation Times of TPC-H Data Sets on Different Cluster Sizes . . . . . 192
18.9. Distribution of Monomial and Orthogonal Coefficients for Degree 0 and 1 193
18.10. The Most Likely Approximating Polynomial for Mondays During the Lecture Period . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 194
18.11. Most Accessed Websites in Stud.IP Between October 24, 2008 and June 10, 2009 per Day . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 196

# Part I.

# Introduction

# 1. Introduction

The amount of data produced by scientific research and business is growing rapidly. A prominent example are social media sites; the amount of data Facebook collected grew drastically from 15TByte in 2007 to 700TByte in 2010 [217]. With the power to store large amounts of data, the interest for analysis is growing. For large amounts of data shared nothing architectures have emerged as a de facto standard [121, 210]. The massive parallelism in the hardware has led to new programming paradigms in data processing that originate from classical high performance computing such as the MapReduce approach [84]. In general, these approaches exchange fundamental database system qualities for better scalability. Furthermore, systems with good scalability frequently have a very bad utilization of system resources [146, 30]. Core achievement and basis of the success of relational database systems is the simple data model combined with a comprehensible, declarative access. This requires transparent access to the distributed data. The distributed access introduces dependencies which have to be adhered automatically.

As the number of nodes in a cluster increases the configuration overhead is growing, in data centers the administration costs dominate the overall expenses [10]. Therefore, more and more tasks have to be automatized. Due to the ever increasing workloads the adding and removal of nodes is such a task. However, the extension of a distributed database system is usually a manual task, which involves complicated data migration processes. Similarly, the data distribution is often a manual task and database administrators spend a fair amount of their time identifying hot spots and resolving them. Although a lot of research focuses on online schema tuning which improves throughput of a single database node, efficiency of a distributed system benefits the most from global optimizations [192]. For a distributed database system the data allocation has a major impact on the overall performance.

Even though there is a large body of research on allocation in distributed databases, only little work has been done on automatic and especially autonomic allocation. A well known description of the allocation problem was presented by Özsu and Valduriez [169]. In order to cover as much influences as possible the authors propose a very complex model. This model includes the storage, processing and data transmission costs. The allocation is formulated as a minimization problem, based on the knowledge of all request and update costs. Other allocation problem definitions and algorithms follow similar approaches. Obviously, this is interesting from a theoretical point of view, but far to complex for real world applications. In order to allow realistic problem sizes and dynamic environments simplified models are necessary.

## 1.1. Motivation

This thesis aims at an automation of scaling and data layout in distributed database systems on cluster hardware. We focus on small to middle sized appliances with a limited number of nodes, but do not restrict ourselves to this setting. Today, database systems are a core element of most data intensive systems. However, especially in smaller projects and startups there is usually only a limited budget for the data management. This is usually not an issue, as long as the database workload can be processed by a single server database system. With modern hardware, single node database systems can be scaled up to enormous processing powers, but at enormous prices. The current leading system in the TPC Benchmark$^{TM}$C of the Transaction Processing Performance Council has a total system cost of over 30 million dollars[1]. Obviously, this is not affordable for most appliances, therefore, many projects use open source database management systems and of the shelf hardware. In these systems, probably the hardest step is from a single database server to a distributed system. In this thesis, we present our shared nothing approach that enables an automatic scale out from a single backend to a distributed system with good speedup.

In a shared nothing environment the overall system performance is strongly correlated with locality. This is because communication is expensive and will eventually always become the bottleneck [17]. Hence, calculations should be performed locally. In a distributed database system on a shared nothing environment it is therefore preferable to process requests locally. The approach is related to classical high performance computing techniques and is diametrical to the declustering approach in parallel database systems. While declustering aims at an reduced execution time for single requests by employing parallelism at operator level, data locality aims at an increased throughput while maintaining single site execution speed.

Since queries are executed locally in this approach, the database backends can be seen as black boxes. Due to this fact, the configuration of the system is much simpler than the configuration of a parallel system. Therefore, different self-management strategies such as autonomic scaling can be implemented relatively easy. Apart from reducing the management costs for adapting the system to increased requirements, the autonomic up and down scaling can also be used to increase the energy efficiency.

If the necessary data is present read requests can always be executed on a single system. However, write requests have to be propagated on all replicas of the affected data set. Therefore, we present a query-aware allocation strategy that distributes data based on the access history. It aims on minimizing the overhead introduced by redundant updates, while balancing the workload and, thus, maximizing the overall throughput.

[1]Top Ten TPC-C Results - `http://www.tpc.org/tpcc/results/tpcc_perf_results.asp` (last visited 2011-04-15)

## 1.2. Contributions

In this thesis, we propose a model for database systems on cluster hardware. Based on this model, an approach for autonomic scaling and allocation is developed. Furthermore, new methodologies for cluster database system benchmarking are presented. The validity of the approaches is proven based on prototypical implementations that are tested with real life examples and standard benchmarks.

Our contributions are the following:

- A processing model for cluster database systems is proposed. It is the formal basis of our prototypes. The model is reduced to an essential core, which considers the most important factors of distributed database systems. Through the reduction of influencing values all factors can be determined automatically.
- We show an approach for autonomic scaling of cluster database systems. Our method is implemented in the Scalileo scaling framework that allows an easy integration of autonomic scaling in distributed systems. We discuss and prove by example, how the efficiency in general and energy efficiency in particular can be improved by autonomic scaling.
- We formalize the allocation problem for our cluster database processing model for read-only and read-write scenarios. We sketch the NP-hardness of both cases and present an optimal computation and heuristic algorithm for each. The algorithms feature an integrated approach for partitioning, replication and allocation. The prototypical implementations show a considerably improved performance compared to full replication. For highly dynamic environments a periodic allocation strategy is shown that exploits reoccurring changes in the workload.
- We propose new methodologies for database benchmarking. A new data generation paradigm is presented. It allows parallel generation of relational data with linear speedups. We explain how different data dependencies can be generated by exploiting the determinism in pseudo random number generation. Furthermore, a new generator for adaptive workloads is presented. The underlying model allows a representation of workload variances by polynomials.

## 1.3. Overview

This thesis is organized in five parts. The first part gives an introduction to the subject. In the next chapter we will give an overview on the formalisms used throughout the thesis. Chapter 3 explains fundamental concepts in relational theory and introduces the cluster database system architecture.

Part II deals with the autonomic scaling of cluster database systems. Chapter 4 explains the mechanisms and prerequisites of autonomic and efficient scaling of distributed systems. In chapter 5 the Scalileo framework – a generic scaling framework – is presented. Chapter

6 explains how the Scalileo framework can be used to build an autonomic self-scaling cluster database system and shows an evaluation of the resulting system.

In part III, different allocation strategies for cluster database systems are explored. In chapter 7 the foundations of distributed data layouts are presented. In chapter 8 related work on allocation strategies is presented. Chapter 9 gives an introduction on the procedures of automatic workload aware allocation. In chapter 10 different approaches for the classification of database requests are presented. Chapter 11 presents an allocation strategy for cluster database systems with read only workloads and chapter 12 extends this definitions for workloads with read and write requests. In chapter 13 both algorithms are extended to ensure high availability. Chapter 14 explains how the allocation can be implemented into a existing configuration with minimum costs. This part concludes with an evaluation of the allocation algorithms in capter 15 and a summary in chapter 16.

Part IV discusses benchmarking of cluster database systems and large, dynamic database systems. Chapter 17 presents standard benchmarks, which were used in this thesis and presents a new benchmark that was defined as part of the thesis. In chapter 18 new methodologies for data generation and workload generation for benchmarking large scale, dynamic systems are presented.

The thesis concludes in part V with an outlook on ongoing and future work.

# 2. Preliminaries

In this chapter we will introduce definitions and formalisms used throughout the thesis. To simplify the understanding of the formulas we use a consistent notation.

## 2.1. Set

We use the term set for a collection of not set-valued, distinguishable objects. Sets are always represented by a single capital letter, e.g. $S$. The elements of a set are represented by a single lowercase letter. We usually use a the according lowercase letter of a sets name to express an arbitrary element of a set, e.g. $s \in S$. To distinguish the elements we use indices, so if $S$ has $n$ elements we write $s_1, s_2, \ldots, s_n \in S$. The cardinality of $S$ is expressed by $|S| = n$. If we want to define a set we do so either by enumerating the elements, e.g. $S = \{s_1, s_2, \ldots, s_n\}$, or by a property of its elements over an other set, e.g. $S = \{s|\text{property of } s\}$.

Special sets used are the natural numbers $\mathbb{N}$, integers $\mathbb{Z}$, and the real numbers $\mathbb{R}$. The the positive real numbers are represented by $\mathbb{R}^+$. Intervals are defined as subsets of $\mathbb{R}$. We use the following short form, $S = ]a, b]$ is the set $S = \{x \in \mathbb{R}|a < x \leq b\}$. Integer intervals are defined by their endpoints like the enumeration of sets, e.g. $I = \{3, \ldots, 5\}$.

## 2.2. Function

We usually define a function $f$ by first giving its domain and codomain, for example $f : A \to B$ indicates that $f$ assigns elements of its domain $A$ to elements of its codomain $B$. A single assignment is written $f(a) = b$, this means that $f$ assigns $a \in A$ to $b \in B$. The definition of the function is then given textual, in a set notation or by constraints.

In order to reduce the number of definitions, we often overload function definitions. For example, the function $f$ from above might also be used for set-valued objects: $f(A') = B'$, with $A' \subseteq A$ and $B' \subseteq B$. The implicit definition of this function would be $f(A') = \{b \in B|\exists a \in A' : f(a) = b\} = B'$.

## 2.3. Family of Sets

A family of sets is a set of subsets of a given set. We represent a family of sets of a set $S$ with a calligraphic capital letter, e.g. $\mathcal{S}$. A special family of sets is the power set, i.e. the set of all subsets of a set. The power set of $S$, is represented by a capital calligraphic letter $\mathcal{P}(S)$. Any family of sets $\mathcal{S}$ of a set $S$ is always subset of the power set of $S$, $\mathcal{S} \subseteq \mathcal{P}(S)$.

A family of sets $\mathcal{S}$ of set $S$ can also be defined as function $\mathcal{S} : S \to \mathcal{P}(S)$. Thus we call $S$ the domain of $\mathcal{S}$.

In general set-valued objects are represented by capital letters. If the elements are also set-valued, we use calligraphic letters or in cases of ambiguity Gothic type.

## 2.4. Multiset

A multiset is a collection of objects and their occurrences. It can be defined over a set $S$ as a pair $\langle S, f \rangle$, where $f : S \to \mathbb{N}$ is a function that assigns each element in $S$ the number of its occurrences in the multiset (we adopt the multiset formulation of Syropoulos [214]). We call $f$ the characteristic function and $S$ the domain of a multiset. We denote a multiset with a calligraphic capital letter, e.g. $\mathcal{M}$. The support of a multiset $\mathcal{M} = \langle S, f \rangle$ is a subset $B$ of $S$, with $B = \{s \in S | f(s) > 0\}$. The cardinality of a multiset $\mathcal{M} = \langle S, f \rangle$ is $|\mathcal{M}| = \sum_{s \in S} f(s)$. The set of all multisets with support $B$ is denoted as $\mathcal{P}^B$.

## 2.5. Sequence

A sequence is a finite or infinite list of objects with an ordering. We write sequences as a single capital letter, e.g. $S$. A sequence can be defined directly, e.g. $S = (1, 4, 5, 6, 7)$ or as a function $S : \mathbb{N} \to C$, where the codomain $C$ is the set of all elements occurring in the sequence. Finite sequences with $n$ elements are also called $n$-tuples.

## 2.6. Graph

A graph $G$ is a tuple of vertices and edges, $G = (V, E)$. Each edge $e \in E$ connects two vertices $u, v \in V$, therefore $E \subseteq V \times V$. A graph is directed, if $e = (u, v) \neq e' = (v, u)$. The edges of a graph can be weighted, then there has to be a weighting function with the following definition:

$$weight : E \to \mathbb{R} \tag{2.1}$$

A graph is said to be complete, if every node is connected with every other node. In a directed graph this requires a two edges to every node, one leading in the vertex and one leading out of the vertex. A special form of graphs are bipartite graphs. In bipartite graphs the set of vertices can be separated in two disjunct sets, $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$, with the property, that there is no edge between two vertices within the same set. A bipartite graph is said to be complete, if each vertex in $V_1$ is connected to every vertex in $V_2$.

# 3. Database, Database System and Database Management System

In the following chapter, we will give a short introduction to relational database systems. This will also be used to introduce the terminology used in the rest of the thesis. Often the terms database (DB), database system (DBS) and database management system (DBMS) are used interchangeably. However, originally they stand for different entities. A database is an organized collection of data, which is typically stored in a database system. To manage and process the data a set of computer programs is used; this piece of software is called the database management system. So the database system is a computer system which stores one or more databases and runs a database management system to give access to the database. This relation is shown in figure 3.1.

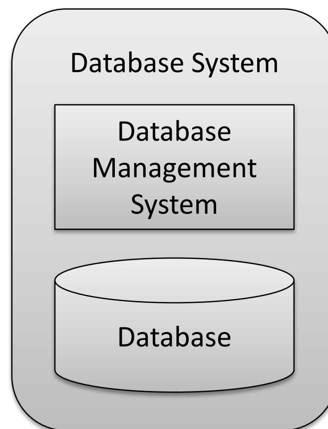



Figure 3.1.: Database System Overview

Following the ANSI/SPARC reference design there are three views of data in a database (see figure 3.2) [218]. The *internal view* for the database management system, the *logical view*, containing data model and the *external view* which gives access to the users and applications. For each of these a schema is defined. The internal schema defines the physical representation of the data, this is how and where relations are stored and which indexes are defined. The logical schema contains an integrated model of all relations of the database and their interconnections. The external views define how data can be accessed by the user.

The internal data representation is the duty of the DBMS. It specifies how the data is

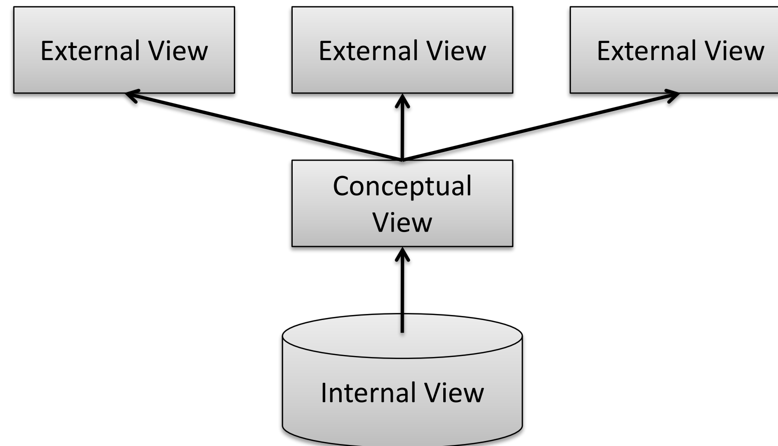



Figure 3.2.: ANSI/SPARC Reference Model

stored and the data structures used to manage and access the data. We will give some details of the internal representation in the next section.

The external views define which data may be accessed by the users. In general the user will not have access to all data stored in the database. Furthermore, it is unusual that all users have access to the same data in the database, e.g. most database systems have an internal catalog of meta data which is not accessible for regular users. This user management is either carried out by the database management system or the application logic on top of the database system.

The logical view is an integrated representation of the data in the database. The representation conforms to a certain specification or *schema*. This schema is usually defined in an application independent data model. There are several structured data models, such as the object-oriented model, the deductive model, and the entity-relationship-model. The most common model is the relational model and its derivatives. As this model is used in the rest of this thesis, we will give an introduction in the following section. After that, we will give a short overview of the architecture of a DBMS, as far as it is relevant to this thesis. Finally, we will introduce distributed and cluster database systems.

## 3.1. Relational Model

The relational model was introduced by Edgar F. Codd in 1970 [69]. In contrast to the models that were used before, such as the network model or the hierarchical model, it is set-oriented. This means that data within the model is organized and processed in sets. This gives the model a very simple structure, compared to the earlier record oriented models.

In the relational model the data is organized in *relations*. Formally defined, a relation $R$ is a subset of the Cartesian product of the domains $D_i$ of its fields.

$$R \subseteq D_1 \times D_2 \times \cdots \times D_n \tag{3.1}$$

Common domains in a database include integers, character arrays and floating point numbers. A single entry of a relation is called tuple. A tuple $t \in R$ corresponding to the definition above consists of $n$ values or *attributes* from the domains $D_1, D_2, \ldots, D_n$. A typical example of a relation is the following relation $ORDERS$

| ORDERS | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 1 | Screws | Mike | 2010-10-11 |
| 2 | Nails | Andy | 2010-11-22 |
| 3 | Hammer | Chris | 2010-11-22 |
| 4 | Nuts | Mike | 2010-12-05 |
| ... | ... | ... | ... |

An example of a tuple is (2,'Nails','Andy','2010-11-22'). Since a relation is similar to a *table*, this term can be used as an alternative. In this context, a tuple is called *row* and an attribute *column*. In the example above the relation has the domain *integer* × *string* × *string* × *date*[1]. This is usually specified in the following manner:

$$ORDERS : \{[\text{ID : integer, ITEM : string, CUSTOMER : string, DATE : date}]\} \tag{3.2}$$

The structure of the relation is also called the *schema*. This representation has the advantage that the parts of the relation are named and can be referenced. We will use the following notion for the $ID$ of $ORDERS$, $ORDERS.ID$, if the attribute name is unambiguous, we will omit the name of the relation. In most of the examples in the rest of this thesis the domains or data types are not important and will therefore be omitted. It should be noted that the term relation is used in the context of an instance as well as the schema. So a relation is a set of tuples and also a set of attributes. We will not formally distinguish these meanings, since they are clear in the context.

In the original relational model all entries in a relation had to be unique, so there were no duplicates allowed. In modern systems this is not always enforced, and if uniqueness is necessary it is often realized by an artificial identifier, such as a row number. A set of attributes that enables a tuple to be identified is called a *key*. Often, there are multiple different keys for a relation; in this case one key is chosen to be the *primary key*. In the example this could be the *ID*. Keys are used to define interrelations between relations. Consider the following relations $CUSTOMERS$, which further describes the customers referenced in the table $ORDERS$:

[1] We use the ISO 8601 international standard date form in the following.

| CUSTOMERS | | | |
|---|---|---|---|
| NICK | FNAME | LNAME | CITY |
| Chris | Christian | Summers | Los Angeles |
| Mike | Michael | Smith | San Francisco |
| Andy | Andrew | Michell | Santa Barbara |
| ... | ... | ... | ... |

The attribute $NICK$ is underlined to indicate, that it is the primary key for the relation. To establish the interrelation between orders and customers in the example, the primary key of $CUSTOMERS$ is an attribute in $ORDERS$. This is called a *foreign key*. To find all data about a customer of a selected $ORDERS$ tuple the tuple in $CUSTOMERS$ has to be found where $ORDERS.CUSTOMER = CUSTOMERS.NICK$. There are several ways to describe this form of data retrieval for the relational model, most common are the *relational algebra* and *relational calculus*. The difference between the two is that the relational algebra describes a procedural way to retrieve the data, while the relational calculus is a declarative description of the data. For safe queries, i.e. queries that are domain independent, both models have equal expressiveness. In the following we will describe the operators of relational algebra and give a definition in the tuple relational calculus. The tuple relational calculus is a set based description of a relational query. We use square brackets to modify tuples in the relational tuple calculus.

### 3.1.1. Relational Algebra

Above, we have described how to define and represent data in the relational model. In this section, we will describe how to query relational data using relational algebra, as far as it is used in this thesis.

To extract tuples that follow a certain form from a relation *selection* is used. It is an unary operator, which has a relation as input and returns a relation. All tuples in the input are tested against a given predicate. An example with the tuple calculus formulation is:

$$\sigma_{CUSTOMER='Mike'}(ORDERS) = \{t | t \in ORDERS \wedge t.CUSTOMER =' Mike'\} \quad (3.3)$$

The result of this selection is again a relation:

| $\sigma_{CUSTOMER='Mike'}(ORDERS)$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 1 | Screws | Mike | 2010-10-11 |
| 4 | Nuts | Mike | 2010-12-05 |

In general, the predicate can be a logical formula. The selection reduces a relation to the required tuples. To reduce a relation to the required attributes, the *projection*

is defined. It is again a unary operator that has a relation as input and output. The projection reduces the set of attributes to the set specified:

$$\pi_{ITEM,CUSTOMER}(ORDERS) = \{[t.ITEM, t.CUSTOMER] | t \in ORDERS\} \quad (3.4)$$

$[t.ITEM, t.CUSTOMER]$ defines a tuple with the two attributes $ORDERS.ITEM$ and $ORDERS.CUSTOMER$. The result of the projection above is:

| $\pi_{ITEM,CUSTOMER}(ORDERS)$ | |
|---|---|
| ITEM | CUSTOMER |
| Screws | Mike |
| Nails | Andy |
| Hammer | Chris |
| Nuts | Mike |

There are several binary operators that enable relations to be combined. If two relations with identical attributes are combined, set operators such as union, intersection and difference can be used. More important are the *Cartesian product* and the *join*. The Cartesian product in relational algebra calculates all possible combinations of the tuples of two relations:

$$\begin{aligned} &ORDERS \times CUSTOMERS = \\ &\quad \{[s.ID, s.ITEM, s.CUSTOMER, s.DATE, \\ &\quad t.NICK, t.FNAME, t.LNAME, t.CITY] | \\ &\quad s \in ORDERS \land t \in CUSTOMERS\} \end{aligned} \quad (3.5)$$

An excerpt of the solution for the relations above is:

| $ORDERS \times CUSTOMERS$ | | | | | | | |
|---|---|---|---|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE | NICK | FNAME | LNAME | CITY |
| 1 | Screws | Mike | 2010-10-11 | Chris | Christian | Summers | Los Angeles |
| 1 | Screws | Mike | 2010-10-11 | Mike | Michael | Smith | San Francisco |
| 1 | Screws | Mike | 2010-10-11 | Andy | Andrew | Michell | Santa Barbara |
| 2 | Nails | Andy | 2010-11-22 | Chris | Christian | Summers | Los Angeles |
| 2 | Nails | Andy | 2010-11-22 | Mike | Michael | Smith | San Francisco |
| 2 | Nails | Andy | 2010-11-22 | Andy | Andrew | Michell | Santa Barbara |
| ... | ... | ... | ... | ... | ... | ... | ... |

The join enables pairs of tuples which satisfy a certain predicate to be selected. In the following example an equijoin, i.e. a join that tests for equality, is shown:

$$\begin{aligned}
&ORDERS \bowtie_{CUSTOMER=NICK} CUSTOMERS = \\
&\quad \{[s.ID, s.ITEM, s.CUSTOMER, s.DATE, \\
&\quad t.NICK, t.FNAME, t.LNAME, t.CITY] | \\
&\quad s \in ORDERS \land t \in CUSTOMERS \land s.CUSTOMER = t.NICK\}
\end{aligned} \tag{3.6}$$

The result for the relations above:

| $ORDERS \bowtie_{CUSTOMER=NICK} CUSTOMERS$ | | | | | | | |
|---|---|---|---|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE | NICK | FNAME | LNAME | CITY |
| 1 | Screws | Mike | 2010-10-11 | Mike | Michael | Smith | San Francisco |
| 2 | Nails | Andy | 2010-11-22 | Andy | Andrew | Michell | Santa Barbara |
| 3 | Hammer | Chris | 2010-11-22 | Chris | Christian | Summers | Los Angeles |
| 4 | Nuts | Mike | 2010-12-05 | Mike | Michael | Smith | San Francisco |

Using combinations of the operators, more complex queries can be built. For example: Find all customers who ordered in November 2010 and give their first and last name:

$$\begin{aligned}
&\pi_{FNAME,LNAME}(CUSTOMERS \bowtie_{NICK=CUSTOMER} \\
&\quad (\sigma_{DATE \geq 2010-11-01 \land DATE \leq 2010-11-30}(ORDERS)))
\end{aligned} \tag{3.7}$$

In order to increase the readability of such queries, they are often presented as an operator tree as in figure 3.3. A tree like structure is also used by several database management systems for query processing.

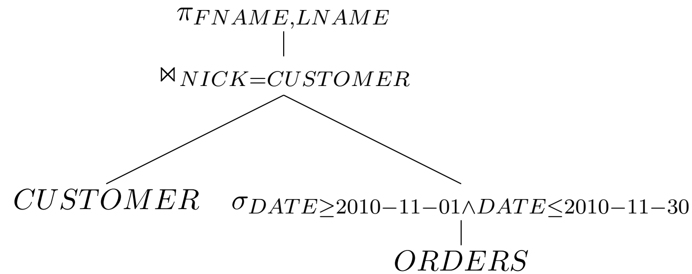



Figure 3.3.: Algebraic Operator Tree

As can be seen in this example, expressions in relational algebra can get cumbersome easily. Therefore other languages were designed which allow simpler query specifications. The most common language today is SQL.

### 3.1.2. SQL

*SQL*, sometimes also referred to as structured query language, is a declarative data definition, data manipulation and data query language. It is supported by most database

management systems. In this section, we will only discuss basic query elements of the SQL 92 standard [1]. We will limit the discussion to simple SQL queries. These have the general form called *select-project-join*. The query in equation 3.7, can be expressed in SQL as follows:

Listing 3.1: Select-Project-Join SQL Query

```
select FNAME, LNAME
  from ORDERS, CUSTOMERS
 where CUSTOMER = NICK
   and DATE >= date ('2010-11-01')
   and DATE <= date ('2010-11-30')
```

SQL queries start with the *select*-clause, which specifies the attributes in the output relation. This is similar to a projection in relational algebra, although the select clause is more powerful than a simple selection. In the second row the *from*-clause can be seen; here all input relations are specified. Finally, in the *where*-clause, constraints that the result tuples have to satisfy are specified. In the example, these correspond to predicates of a selection in relational algebra. In the last two rows, a date function can be seen, it converts a given string to a date. Joins are usually specified only indirectly, by specifying multiple relations in the from-clause and according join predicates in the where-clause. In the example this can be seen for $ORDERS$ and $CUSTOMERS$ and their attributes $CUSTOMER$ and $NICK$.

Even though SQL is the most common way to access a database, most users of a database system will never encounter an actual SQL query. Usually, this is hidden by a form or application. As can be seen in the example above, the SQL queries are declarative and do not give any instructions on how to process the data. Therefore, the DBMS has to translate them into database operations. In the following section, we will explain the DBMS architecture and query processing.

## 3.2. Architecture of a Database Management System

Database management systems are highly complex software systems. Therefore, a layered architecture was proposed early by several authors [116, 113, 201]. We review the 5-layered model by Härder and Reuter [132]. Each of the layers equals a step in the transformation from the abstract query language to the physical file access (see figure 3.4). Even though it is a model for the architecture of a database management system, most current database management systems implement the most of the transformations in such a way. In the following we will explain the different layers and transformation steps.

**Data system: nonprocedural or algebraic access** As mentioned above, most database systems feature the relational model and SQL access. This is called the set-oriented interface, it allows a declarative description of the data and the queries. The data system layer has to transform the declarative access to the navigational access.

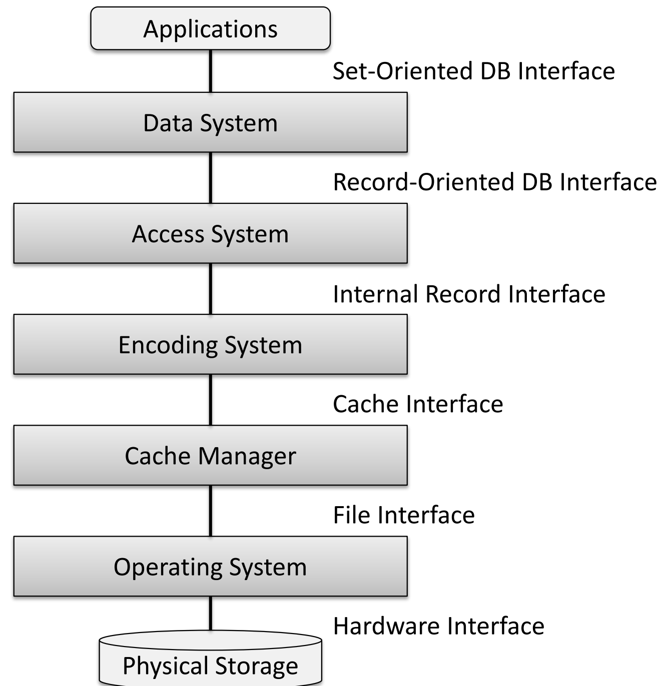



Figure 3.4.: The 5 Layer DBMS Architecture Model

Internally, it features a data model that is similar to the relational algebra, it has logical operators that are abstracted from the actual implementation. On this layer, the queries are optimized and translated to the record-oriented interface of the access system. A record is the internal representation of a tuple.

**Access system: record-oriented, navigational access** The record-oriented interface enables internal representations of the relations to be addressed; this can be various access paths such as indexes or scans. The access is typed. On this layer, operations such as sorting and joining are also implemented. The access system translates the logical access to concrete access of typeless records on the internal record interface, provided by the encoding level.

**Encoding system: record and access path management** The encoding system provides a virtual linear address space on all records. It implements operations on the physical access paths, such as B-tree operations. Internally, the data is managed in fixed sized pages, which store records. A page is a continuous data block with a linear address space. The page access is controlled by the cache management.

**Cache manager: propagation control** The cache manager features a page oriented interface. The pages are mapped to the block oriented interface of a file. In order to minimize disk reads and writes a cache manages the pages currently in the memory.

**Operating system: file management** The lowest layer is concerned with the file management. In early database management systems this functionality was also implemented by the DBMS. Today, this is usually done by the operating system's file management.

To illustrate the query processing, we will review the transformations on the query in listing 3.1. This is to be understood as a generic example, specific systems will differ in the intermediate results and data structures. When the SQL query is sent to the DBMS the data system translates it to a logical operator tree. As mentioned above, this is similar to relational algebra. The initial tree is a direct translation without any optimizations (see figure 3.5).

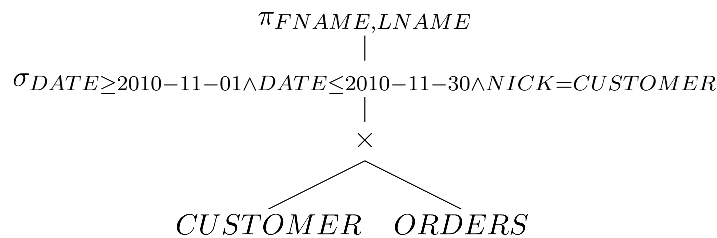



Figure 3.5.: Logical Operator Tree

The next step is the logical optimization of the operator tree. Typical examples are the replacing of Cartesian products by joins and pushing down selections. A possible result of this can be seen in figure 3.3. In the next step, the logical operators will be replaced by physical operators and access paths. Usually there are multiple possible physical operator trees. In the example the choice of the access paths and join order can lead to very different operator trees. A possible result can be seen in figure 3.6

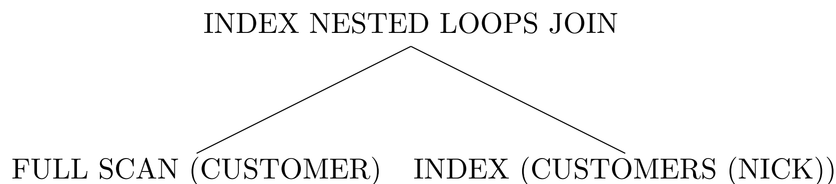



Figure 3.6.: Physical Operator Tree

The physical operator tree is sent to the access system. The access system executes the operators and will generate record accesses, index accesses, etc. Often the operators are implemented according to the *iterator pattern*. Each operator has the methods *open*,*next* and *close*:

**open** Initializes the operator. For example, the access to the relation is opened or auxiliary data structures are initialized.

**next** Generates the next tuple. This can be a single access to the relation, or finding a joining tuple.

**close** Closes the operator. Auxiliary data structures are released, access streams are closed.

In the example in figure 3.6, three operators are shown. When the query processing is started, the JOIN operator is opened, it will then open the SCAN and the INDEX operators. When the *next* method of the JOIN operator is invoked, it will repeatedly invoke the *next* methods of the SCAN and INDEX operators, until it finds the first matching pair of tuples. When the *close* method is called on the JOIN operator, it will also close its successors.

The accesses generated are sent to the encoding system. The encoding system translates the virtual addresses to physical page addresses and requests the relevant pages from the cache manager. The records stored in the pages are then given to the access system.

The cache manager stores frequently used pages in the memory to reduce the number of disk reads and writes. As in the example above, most database queries access small continuous parts in the database. A common caching strategy is the *least recently used* algorithm (LRU). In the LRU algorithm, the page in the cache that has not been used for the longest amount of time will be stored to disk and replaced by a newly required page. The page size is usually aligned with the block size of the disk, so that a page size is an integer multiple of the disk block size.

Finally, the cache manager will send read and write commands to the file system. It is possible to optimize the read and write sequences according to physical characteristics of the disk. For example, on traditional hardware it is much faster to read bulk sequences than to randomly search single blocks.

## 3.3. Distributed Database Systems

In the section above, we discussed the architecture of a single DBMS. However, since data sizes and access workloads often exceed the capabilities of a single system, distributed systems are used. In this section, we will discuss several distributed architectures and give reasons for the use of the cluster database system architecture.

There are several motivations to use distributed systems. Increasing utilization and therefore efficiency was probably the earliest reason. Interestingly enough this was also the reason that led to the development of the ARPAnet, which ultimately was the foundation of the Internet. To increase the utilization of expensive, powerful mainframe computers, interactive, remote terminals were developed. This approach is called a client server model (see figure 3.7). The server provides a service or resource that is requested by the client. For the communication between client and server a network is used. A typical example of a client server system is the combination of web server and web browser. Today, most database systems are server systems and the application programs access the database as a client. This enables the use of dedicated hardware exclusively for the database system.

This in turn enables the system to process a higher workload than a system running multiple services.

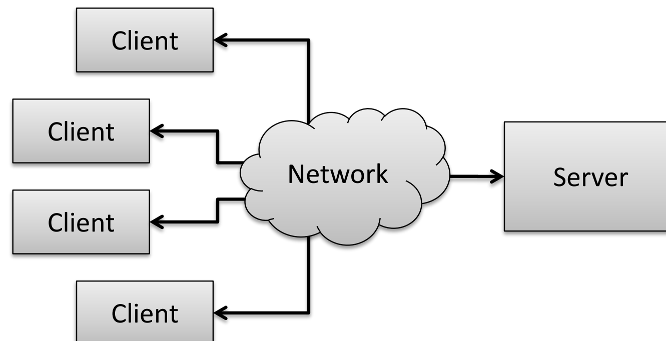



Figure 3.7.: Client Server Architecture

However, in many cases a single system has not enough capacity to match the demand required. Therefore, the database system has to be run on distributed hardware. Several forms of distributed database systems exist. One possibility of classification is their degree of integration:

**Multi database system** Multi database systems are very loosely coupled, autonomous database systems. Usually, they hold different, local databases, which do not share a common schema. Such systems often result from the combination of different applications and are therefore very heterogeneous. Typical examples are information systems for hotel booking.

**Federated database system** Federated database systems are more tightly coupled than multi database systems. They may also have different schemata and local data, but they share a global database. This reduces the autonomy of the single systems: the schema of the globally visible data cannot be changed without coordination with the other participating systems. Federated database systems are again the result of an integration of several systems.

**Integrated distributed database system** Usually, integrated distributed database systems are implemented by a distributed DBMS. The system appears as a single DBS. The global schema is a union of the local schemata. These systems are initially designed to be distributed, with the focus on high performance, fault tolerance and decentralization. They often have a very homogeneous architecture.

The three classes of systems offer very different challenges. However, if the performance of a single database system has to be optimized, obviously especially integrated distributed DBMSs are interesting. Performance gains are achieved by parallel processing of queries. Two forms of parallel query processing can be distinguished, inter-query parallelism and

intra-query parallelism. On a system with inter-query parallelism multiple queries can be processed in parallel. If a system features intra-query parallelism, single queries are processed in parallel. While inter-query parallelism increases the throughput, i.e. the number of queries per time unit, intra-query parallelism decreases the answer time, i.e. the amount of time until a query is processed (see also section 3.4). Usually, these systems are implemented on a high performance computer system. Traditionally, the following architectures are differentiated:

**Shared-everything** All processors in the system share the same main memory and disks. Basically, the DBMS on a shared-everything architecture works very similarly to a single system DBMS.

**Shared-disk** Every processor has its own main memory, but all share the same disks. This architecture allows all processors to access all stored data. This eases the administrative overhead for the data distribution, but increases the difficulties of locking management.

**Shared-nothing** In a shared-nothing system, all processors have separate RAM and disk. Typically, this is a set of independent machines that are connected over a high speed interconnect.

All three architectures are used in professional database systems today. However, shared-everything architectures are very expensive and do not scale limitlessly. Shared-disk systems were especially popularized by Oracle RAC; they scale well with little administrative cost. But in very large scale systems, the more complex locking procedures can become a bottleneck [121]. Shared-nothing systems are frequently made up from *clusters* of off-the-shelf hardware [29, 37], are cheap and have been shown to scale to very large numbers of nodes. For example, Google used a 4000 nodes cluster to sort 1 petabyte of data[2]. Typically, each node runs an independent DBMS that manages only a share of the database. The complete database is *partitioned* across the nodes [208]. Apart from the superior scalability, these system can easily be made failsafe using replication of data. In large scale systems disk and server failures frequently happen, therefore shared-nothing systems have to be able to tolerate loss of single disks or complete servers. Since current processors usually have multiple cores, the independent DBMSs in a shared-nothing system are run on shared-everything architectures. Database systems that are implemented on shared nothing clusters are called *cluster database systems* (CDBS). In contrast to shared-everything and shared-disk systems, shared-nothing systems rely on sophisticated data partitioning. This is usually the work of highly qualified database administrators, whose wages often dominate the maintenance of a database system.

The architecture of a cluster database system can be seen in figure 3.8. It is usually three-tiered and consists of the presentation tier (client), the logic tier (controller) and the data tier (backend). This design is a variant of the client server model, in which

[2]The official Google Blog - `http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html` (last visited 2011-04-15)

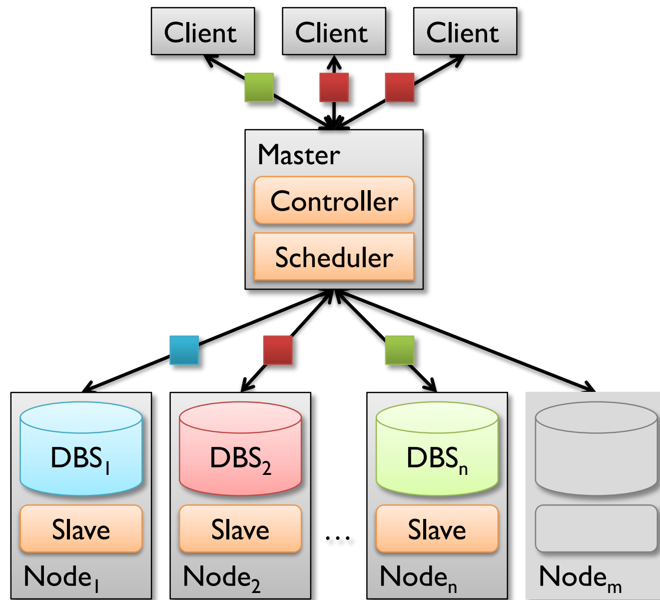



Figure 3.8.: Cluster Database System Architecture

the server component is split up again. Usually the controller is a relatively lightweight middleware system that distributes the incoming queries to the backends. The backends are fully-fledged database systems that each manage either a full replica or a a part of the global database. Although it would be possible to move some of the DBMS functionality completely from the backend to the controller, it is usually more efficient to use a complete DBMS on the backend. In this way the backend DBMSs do not share any resources and therefore also reflect the shared-nothing architecture. This design scales well in the number of nodes if the single queries can be processed by a single backend. However, for large numbers of nodes the controller can become a bottleneck. In this case it is possible to use a hierarchy of controllers that manage subsets of the backends. In the rest of the thesis we will only consider single controller systems. In the next section, we will introduce a simplified processing model for cluster database systems that allows automatic partitioning and allocation.

## 3.4. CDBS Processing Model

To increase the speed of a DBMS on a distributed system, some form of parallelism has to be employed. As mentioned above, based on the processing of single queries, two forms can be distinguished. A system with intra-query parallelism splits up a single query into multiple query fragments and then fragments can be processed in parallel. This is illustrated in figure 3.9. Using inter-query parallelism multiple queries can be processed in parallel. Employing intra-query parallelism can reduce the processing time of a single

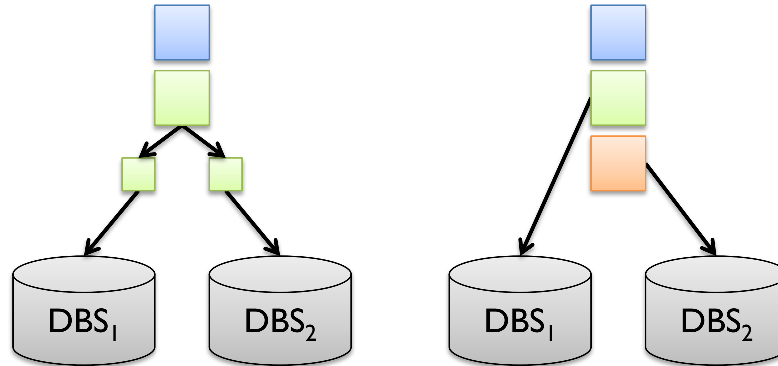



Figure 3.9.: Intra-Query Parallelism (left) vs. Inter-Query Parallelism (right)

query, while inter-query parallelism only increases the throughput. However, intra-query parallelism introduces dependencies between nodes which will lead to an increase of network communication. Apart from decision support and scientific database applications, most queries are relatively small, which means they can be processed in reasonable time on a single system. With the advent of multi-core processors, many DBMSs feature intra-query parallelism, which will speed up the local processing of a query. For this reason, on the cluster level only inter-query parallelism is commonly used. Apart from performance issues distributed query processing can also be used to allow querying distributed data. This is necessary in integrated systems that have no influence on the data distribution.

However, in a CDBS we consider a query as an atomic unit. Each incoming query is processed completely by a single backend. It has to be assured that a backend has all input data for the queries it will process. Since all queries can be processed locally, there is no need for communication between the nodes. Each query can be sent to any backend that has all required data for the query. A common approach in cluster database systems is *full replication*, so each query can be processed by each backend. To achieve a fairly balanced load distribution between the nodes, a simple online strategy can be applied for the query scheduling: the *least pending request first* strategy (LPRF) sends a query to the backend, which at the time has the least queries queued [57]. This strategy is based on the so called *Greedy* algorithm for online load balancing [48, Chap. 12]. Its competitive ratio is 2, which means that in the worst case an optimal offline solution might be at most twice as fast as the Greedy algorithm. In contrast to an online algorithm an offline algorithm has complete knowledge of all events in advance. Only randomized online algorithms may have better competitive ratios than the Greedy algorithm [18].

Obviously, updates cannot be simply processed by a single backend. Instead they have to be executed on each replica of the updated data. The straightforward approach is therefore to send the update to every backend that holds data that will be updated. Combined to the query processing approach above, this is called the *read once / write all* protocol (ROWA) [120, Chap. 2.2]. In order to keep the databases in a consistent state, it is important, that all backends get the updates in the same order. Although there are

more efficient approaches for update synchronization, such as primary copy [20] and lazy replication [145], we will limit the discussion to the ROWA protocol. In the following, we will give some estimations on the performance of a cluster database system. These use a simplified processing model, we will discuss the limitations of the model below in section 3.4.1.

In its most simple form a cluster database system places a full replica of the managed database on each backend. This means that each query can be sent to any backend. If the workload consists only of read requests, then the throughput of such a system will increase linearly with the number of nodes. The throughput can be calculated with the following formula:

$$\text{throughput} = \frac{\#\text{queries}}{\text{processing time}} \tag{3.8}$$

Obviously, the throughput is also dependent on the utilization of the systems. Therefore, the maximum throughput which will be reached at the peak performance of the system is searched. Based on the throughput of two systems with different scale the speedup can be computed:

$$\text{speedup} = \frac{\text{throughput}_1}{\text{throughput}_2} \tag{3.9}$$

Usually, the interest lies in the speedup of a clustered database system compared to a single node system. If the number of queries is the same in all tests, the speedup is only dependent on the processing time of the queries. Theoretically, the maximum throughput in a homogeneous system with only read requests should be proportional to the number of nodes. If no additional overhead is introduced, it should be equal to the number of nodes:

$$\text{speedup}_{\text{opt}} = \#\text{nodes} \tag{3.10}$$

However, in a fully replicated system updates have to be processed on each backend. Therefore, the throughput of the system will decrease with the number of updates. The throughput can be defined as the amount of queries and updates processed per time unit.

$$\text{throughput} = \frac{\#\text{queries} + \#\text{updates}}{\text{processing time}} \tag{3.11}$$

On a system with multiple nodes, the number of updates processed by the system increases with the number of nodes:

$$\text{processed updates} = \#\text{updates} \times \#\text{nodes} \tag{3.12}$$

On a homogeneous, fully replicated system, each update will take the same amount of time on each node. Therefore, it takes the same time as if the updates are processed on a single node, while the queries can be distributed on the nodes. So in theory the

processing time of the updates is constant, while the processing time of the queries is inversely proportional to the number of nodes:

$$\text{processing time} = \frac{\text{processing time queries}}{\#\ \text{nodes}} + \text{processing time updates} \tag{3.13}$$

Hence, the speedup will decrease with the processing time of the updates. The correlation is defined by Amdahl's law [23]:

$$\text{speedup} = \frac{1}{\frac{\text{parallel}}{\#\text{nodes}} + \text{serial}} \tag{3.14}$$

In this context, *parallel* is the ratio of query processing time and *serial* is the ratio of update processing time. This model has some limitations which will be discussed in the next section. Nevertheless, it allows a closed formal approach to other forms of data allocation and a good throughput prediction.

### 3.4.1. Limitations of the Model

As mentioned above, the model has several limitations. Amdahl's law was criticized for being to pessimistic since it does not take positive effects such as caching into account [112]. Obviously, the same is true for database systems. If single systems frequently process similar queries, they will have increased performance because of positive effects of the cache manager. Especially when the database is partitioned (see chapter 7.1), less data has to be read and hence the query processing is sped up. Furthermore, the model only calculates the maximum speedup; it makes no assumptions about the deviations in the processing speed of the queries. There are several factors that could vary:

- Unbalanced load
- Query dependencies
- System variations

As explained above, we use a greedy scheduling algorithm. Hence, the algorithm can perform differently on different scales of a system. The deviations in the processing times between backends can be very different, especially if very long running queries are scheduled. This results in an unbalanced load, which will decrease the throughput of the system. A further factor which causes variations is interdependencies of queries. If similar queries are processed by a single backend, the optimizer will reuse query plans and the required data will already be cached. Finally, even on a single node system, processing times of a query will vary because of variations of the system load from other processes and concurrent disk reads. Therefore, exact predictions on the speedup are not possible.

Another important aspect of DBMSs that is not considered by the model is the concept of *transactions*. Transactions combine a series of queries and updates to an atomic process.

Transaction management ensures that each transaction is either processed completely or revoked. This allows consistency and correctness of the data. Although all current enterprise DBMSs support transactions, there is a trend in large scale systems away from transactions [213]. This is because of the overhead introduced by distributed transaction management. In the next section, we will give some remarks on how to integrate transaction management into the model presented.

### 3.4.2. Transactions

Transactions are necessary to manage multi-user access, a common example are accesses to bank accounts: If an account is accessed by multiple users in parallel, it has to be guaranteed that concurrent updates do not get lost or corrupt the state of the account. A withdrawal is usually implemented by a query that retrieves the current balance and a subsequent update that sets the new balance. If multiple withdrawals are done in parallel without transactional support, all queries read the same balance and the updates overwrite each other. In this way only the last withdrawal will be persistent, the others are lost.

Transactions follow the *ACID* paradigm [133]: a transaction has to be *atomic* and it has to leave the database in a *consistent* state. If multiple transactions are processed in parallel, they should not interfere with each other; they should be processed in *isolation*. Finally, the effects of a transaction have to be *durable* in the database. There are several techniques to implement transactions in a database system, such as two phase locking and multiversion concurrency control (surveys can be found in [38, 224]). As mentioned above, most current systems support transactions. To implement distributed transactions a two-phase commit protocol (2PC) can be used [106]. In the first phase of the 2PC protocol the master asks all nodes that take part in a transaction if they are ready to commit. If all are ready, the master sends a commit to the nodes in the second phase. If one or more fail or time out, the master sends an abort. This protocol is widely used in distributed transaction systems. More advanced protocols have been proposed to avoid blocking of nodes after failures, but they have shown to be too inefficient for realistic workloads [136].

The SQL 92 standard defines four different levels of isolation for transactions [1]: *Read Uncommitted*, *Read Committed*, *Repeatable Read*, and *Serializable*. These differ in the way how transactions may interfere with each other. In the Read Uncommitted level dirty reads are possible. Hence, a transaction can read data that an other transaction has written before it has committed. In the Serializable level the transactions see only their own data changes as if they are processed one after the other. Serial execution reduces the possibilities for concurrency in the database system, therefore often non standard levels such as *snapshot isolation* are used [35]. These multiversion approaches allow transactions to work on their on version of the database.

In a CDBS four types of transactions can appear. Read-only or update and local or distributed transactions. Local transactions are handled completely by a single local database system. Distributed transactions have to be started on all participating nodes, while reads can be processed on a single node, the updates have to be processed on all

nodes with the relevant data. For full replication this includes all nodes. Basically, the single queries of a transaction are processed in the same way as without transactions. Hence, the throughput only deviates from non-transactional processing if transactions interfere with each other. This is if two transactions require the same data and at least one is an update. This happens only on very rare occasions, Gray estimates a percentage of 0.001 or 0.0001 for online transaction systems [109]. Hence, the resulting decrease of the throughput can be estimated with a constant factor.

Besides traditional 2PC protocols, a CDBS can make use of the transaction ordering. If the transactions are processed on all nodes in the order of their arrival, all nodes always have the same state. Therefore, a simple one-phase commit protocol is sufficient. It can be implemented in the middleware [90, 202] or directly in the backend DBMS [216]. Following this approach, a distributed transaction has to be started on all nodes that have relevant data. If the operations are processed in the same order on all nodes participating, all nodes either succeed or have a conflict. Obviously, some further restrictions have to be considered such as the use of the current system time in a query.

## 3.5. Scientific and Commercial CDBSs

There are many implementations of cluster database systems that use the processing model presented. In this section, we will give an overview of scientific and commercial systems. The list is far from being exhaustive. What is more, many database systems that carry the word cluster in their name have quite a different processing model. These usually derive their name from the underlying hardware platform. A prominent example is Oracle Real Application Cluster (RAC), which has a shared disk architecture [7].

Scientific projects are often implemented in the form of a middleware, using off-the-shelf database management systems. Examples are the C-JDBC project and Ganymed.

### 3.5.1. C-JDBC

The C-JDBC project is a middleware platform for transparent database replication [59]. It provides a Java Database Connectivity (JDBC) driver that allows the distributed system to be accessed as if it were a single database (see figure 3.10). The database backends only have to provide a JDBC driver again. The replication mechanism in C-JDBC is called Redundant Array of Inexpensive Databases (RAIDb) [57], which is an analogy of classical RAID for databases [174]. The C-JDBC driver was the basis for the first prototype of a cluster database system for this thesis, which was presented in [188]. However, it is primarily designed for full replication and does not allow partitioning. Still, the relations of a database can be distributed according to the following RAIDb levels:

**RAIDb-0** RAIDb-0 defines full partitioning. It is similar to the RAID level 0. Each relation in the database is allocated on a single backend. The overall disk usage is equal to a single system. Similar to RAID 0 the *mean time between failure* (MTBF) is inversely

proportional to the number of backends. For $n$ backends, where each has MTBF $t$, the MTBF is $\frac{t}{n}$. Furthermore, the size of the cluster is limited to the number of relations in the database. Since C-JDBC does not support distributed query processing RAIDb-0 can in general only be used for queries with single tables access.

**RAIDb-1** RAIDb-1 defines full replication; each database backend manages a copy of the complete database. A RAIDb-1 system provides highly increased fault tolerance and high read-only query processing speed, at the price of high disk usage. The controller has less work for request dispatching, since read requests can be sent to any backend and write requests have to be sent to all backends according to the ROWA protocol. This level limits the size of the database to the disk capacity of a single backend. For databases with a considerable number of update requests the scalability of a RAIDb-1 system is very limited, as described in section 3.4. RAIDb-1 is the default level for C-JDBC.

**RAIDb-2** The RAIDb-2 level allows partial replication of the relations. To provide fault tolerance, each relation has to be allocated to at least two backends. An example of a RAIDb-2 configuration can be seen in figure 3.10. The controller has to do more work than for RAIDb-1, since it has to decide which backend can answer incoming requests. Since C-JDBC cannot process distributed queries, each incoming query has to be processed by a single backend. To assure the existence of an adequate backend, one backend usually contains the complete database.

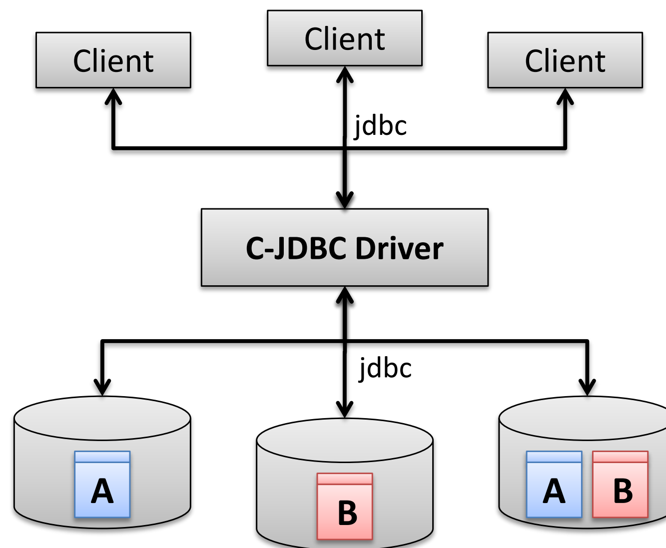



Figure 3.10.: Architecture of a C-JDBC Cluster with RAIDb Level 2

The C-JDBC project was continued as an open-source project in the Sequoia project.

This was then further extended in the Tungsten projects[3] [74], which feature a complete middleware stack for horizontally scaled database systems.

### 3.5.2. Ganymed

The Ganymed project[4] provides a middleware layer that allows replication of databases and transparent access [177, 178]. The system is divided into a master and several satellite database systems, in which the satellites feature read-only access, while write access is processed only on the master node (see figure 3.11). The consistency of the database is guaranteed by snapshot isolation. The system has a very similar processing model to the one presented in section 3.4. It gains speedup by processing each query on a single node. However, to speed up the update propagation, a *lazy* approach is used. This means that updates are not necessarily propagated at their arrival and that the state of the backend databases may temporarily differ. Furthermore, the system allows the dynamic creation of specialized satellite databases. These can be satellites for skyline or keyword searches. To ensure an optimal throughput for write-heavy workloads, the system can automatically scale down to the master.

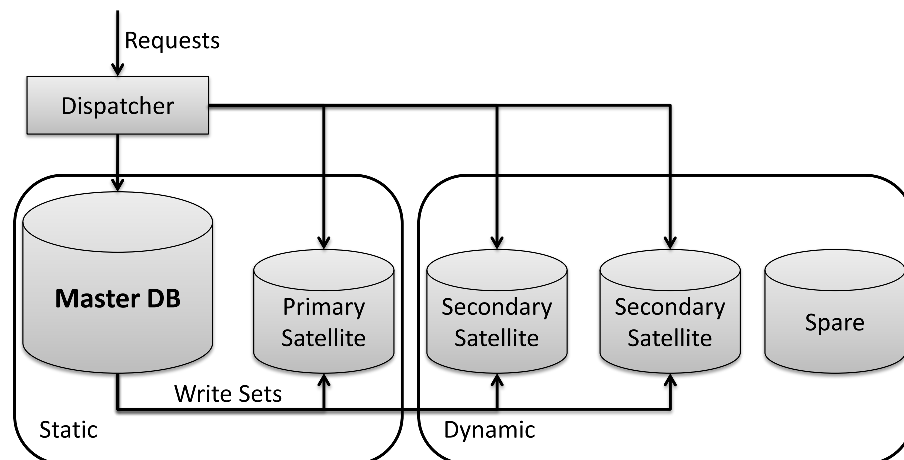



Figure 3.11.: Overview of the Ganymed Architecture

### 3.5.3. MIDDLE-R

The system Middle-R implements a middleware for database replication [155, 173]. The work is based on the Postgres-R system [135], which implemented a similar system within the PostgreSQL DBMS. The system uses group communication to synchronize update requests. This way requests can be directed to any node in the system and there is no need for a central master node. The architecture can be seen in figure 3.12. The system

[3] Tungsten Replicator - `http://code.google.com/p/tungsten-replicator/` (last visited 2011-04-15)
[4] The Ganymed Project - `http://www.ganymed.ethz.ch/` (last visited 2011-04-15)

consists of a queue manager, a database interceptor and a communication manager. The queue manager implements the replication protocols and interacts with the database system using the database interceptor. The database interceptor connects to the DBMS and submits incoming transactions. The communication manager is the queue manager's interface to the group communication system. In order to increase network performance multicast messages are used.

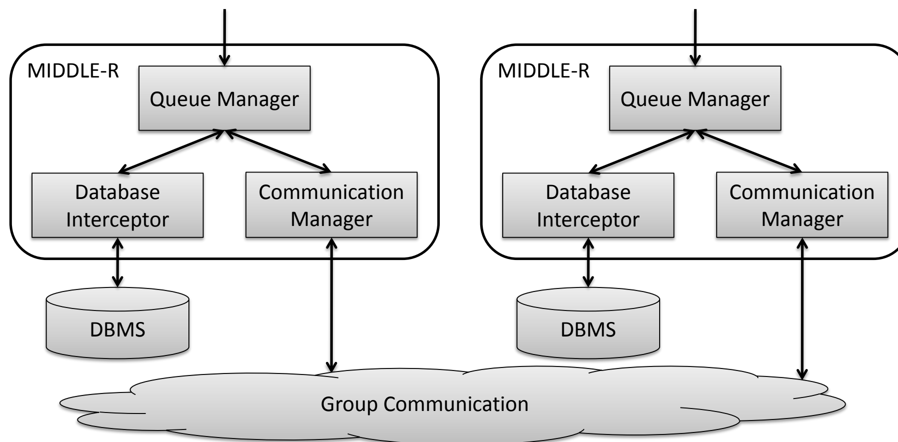



Figure 3.12.: MIDDLE-R Architecture

### 3.5.4. MySQL Cluster

MySQL Cluster is an extension of the open-source MySQL database management system designed for high availability and high performance [167]. It has a shared nothing architecture, which can be seen in figure 3.13. There are three different node types:

**Data node** stores the data,

**Management node** configures and monitors the cluster,

**SQL node** processes SQL queries.

In MySQL cluster, data is usually partitioned and replicated across the cluster. The replication uses a master-slave approach, similar to Ganymed. Management nodes monitor the system and allow re-synchronization in cases of errors. The data nodes store the database and manage the access. Initially, the data was stored only in-memory, but new versions allow disk-based storage. SQL nodes translate SQL queries to low-level API calls to the data nodes. Since the data nodes do not act as complete DBMSs, the system is strictly speaking not a cluster database management system. However, on the data node level it works in a similar way to the processing model presented.

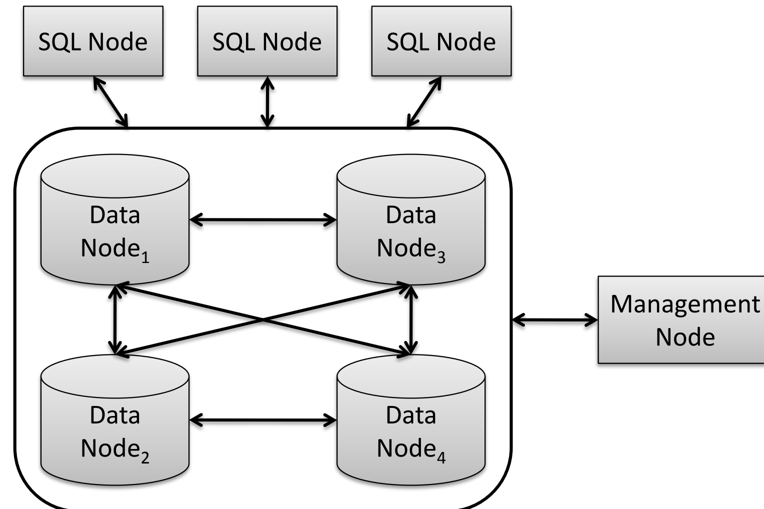



Figure 3.13.: Architecture of the MySQL Cluster

### 3.5.5. NonStop SQL

NonStop SQL is a commercial parallel database management system. It was originally developed by Tandem Computers [92]. Today it is distributed by HP in the Neoview platform [6]. NonStop SQL can achieve linear speedup by horizontally partitioning data across multiple shared nothing nodes and by executing the same query on all nodes in parallel. For the execution of updates, this is equivalent to the processing model presented. For queries however, the system employs intra-query parallelism.

### 3.5.6. DB2

DB2 is a commercial DBMS produced by IBM [67]. Like most business DBMSs, it can be run on shared everything and shared nothing architectures. DB2 has a shared nothing architecture. IBM specified the Distributed Relational Database Architecture (DRDA) standard for the interoperability of distributed databases based on an early version of DB2 [4]. This standard defines four different levels of distributed access to distributed databases:

1. *Remote Request* enables a single request to be sent to a distant server.
2. *Remote Unit of Work* enables a transaction to be sent to a single distant server.
3. *Distributed Unit of Work* enables requests of a transaction to be distributed to different distant servers.
4. *Distributed Request* enables a single request to be distributed to multiple servers.

Recent DB2 versions support all four levels of DRDA. Up to level three, the query processing is similar to the model presented. Level four includes intra-query parallelism.

### 3.5.7. Discussion

The presented systems all have a relation to the CDBS query processing model. In the following tabular the differences can be seen.

| System | Query processing | Update propagation | Synchronization |
|---|---|---|---|
| C-JDBC | Local | Eager | ROWA |
| Ganymed | Local | Lazy | Primary Copy |
| MIDDLE-R | Local | Eager | Group Communication |
| MySQL Cluster | Local | Eager | Primary Copy |
| NonStop SQL | Distributed | Eager | RAWA |
| DB2 | Both | Both | Primary Copy |
| CDBS | Local | Eager | ROWA |

The closest thing to our approach is C-JDBC. For full replication the approaches are identical. Ganymed has an equivalent approach for query processing, however, in contrast to our work it uses a primary copy update mechanism. This reduces delays that stem from synchronization, but makes performance estimations difficult. For our processing model this analysis is easily predictable, as can be seen in section 3.4. The same is true for the MIDDLE-R system, while query processing is equivalent to the CDBS model, updates are synchronized by group communication. This reduces the network traffic, but again limits predictability.

MySQL Cluster, NonStop SQL, and DB2 have a slightly different processing model. MySQL Cluster has a two-staged query processing, in which low level API calls are processed similar to the CDBS model. NonStop SQL is a classical parallel database system, data is fully declustered (see section 8) and every query is processed at all nodes (read-all), similarly updates are processed at all nodes as in the CDBS model (write-all). Our model could be adapted to comply with the parallel model. The enterprise database system DB2 has various configuration possibilities that range from the CDBS model to a parallel processing model.

The major advantage of our approach is the good performance predictability. The determinism in the CDBS approach reduces processing complexity and has been shown to be scalable and adjusted to modern technology trends in recent research [202, 216]

# Part II.

# Scaling

# 4. Scaling Distributed Database Systems

Scalability is of major interest for Internet based applications. Access peaks that overload the application are a financial risk. Therefore, software and hardware systems have to be built to scale. Usually, they are configured to be able to process peaks at any given moment. Obviously, this is very inefficient. Yet, there are various ways to improve efficiency. One reasonable, straight forward approach is to scale applications according to their workload at any particular time.

Scalability of a system involves two abilities: on the one hand, to deal with increased workload and on the other hand, to increase the performance of a system by adding resources [225]. According to Bondi four types of scalability of a software system can be differentiated: *load scalability*, *space scalability*, *space-time scalability*, and *structural scalability* [47]. We will examine the cluster database system architecture according to these types:

**Load scalability** A system that has a stable performance under different amounts of load is said to be load scalable. For database systems this means that it can process requests with equivalent answer times for low, medium or high loads, when the load never exceeds the maximum throughput of the system. For queries in a cluster database system the load scalability is obviously dependent on the load scalability of the backend database systems. Each read-query is scheduled to one backend and the processing of the queries in the middleware is independent of the number of queries in the system. The scheduling of updates is per se not load scalable, since the full replication enforces the middleware to replicate the updates to all backends, reducing the update speed to the speed of a single node system.

**Space scalability** The space requirements of a system in relation to the number of managed elements define its space efficiency. A system is space efficient if its memory and disk requirements increase to an acceptable degree when the number of managed elements increases. In a cluster database system the space efficiency concerning memory is again mostly dependent on the backend database system. The middleware has only minimal space requirements, since it only holds the data dictionary in order to know where to send the queries. The memory requirements of a database system are usually space scalable, since buffer strategies allow the processing of various database sizes with a fixed memory size. The disk space requirements increase linearly according to the number of tuples in the database for a single system. The degree of replication increases the requirements of the cluster database system by a constant factor, which for full replication is the same as the number of backends in the system. Because of this, the disk space scalability is somewhat limited.

**Space-time scalability** A system is said to be space-time scalable if it still functions if the number of managed elements is increased by numbers of magnitude. This scalability is again mostly dependent of the backend database systems. The middleware can manage various numbers of backends, if its limits are reached, a hierarchical middleware structure allows further scalability. For the backend systems, the scalability is again dependent on the scalability of a single system. Usually database systems allow the management of various sizes of databases and are therefore space-time scalable.

**Structural scalability** If a system is not limited in the number of elements it manages by implementation, it has structural scalability. Structural scalability is limited if address spaces limit the number of managed elements to a fixed number. In general the middleware has no restriction on the number of backends therefore it is structurally scalable. Modern database systems usually have flexible addressing schemes that allow astronomical numbers of tuples in a database (current versions of the Oracle Database Server allow up to 8 exabytes, i.e. $2^{60}$ bytes, in a single database [5]).

As shown, the cluster architecture is not scalable in all dimensions. The update load scalability and the disk space scalability are limited. However, these limits stem from the use of full replication. Because of full replication the database has to be stored completely on to all backends and the updates have to be replicated to all backends. Using more advanced allocation strategies the degree of replication can be greatly reduced. This reduces the disk space requirements and increases space scalability. Furthermore, the load scalability for updates is increased. We will give more details on allocation strategies in part III, therefore, we will postpone the treatment of this problem at this point.

A system that has a good scalability should be able to scale. For distributed systems, this can be done in two ways: vertically and horizontally. The two forms are depicted in figure 4.1. To scale a system vertically or *scale* it *up* means increasing the resources of a single system [85]. This can be done by changing the CPU, adding RAM or completely migrating to a more powerful server. Horizontal scaling or *scale out* scalability means increasing the number of nodes in a distributed system. In a CDBS this means adding a new backend node to the cluster.

As obvious in figure 4.1 vertical scaling maintains the original system architecture, while horizontal scaling changes the software architecture by increasing the degree of distribution of a system [225]. In the extreme case a vertically scaled system can still work as a single integrated system, while a horizontally scaled system with equal performance already has tens or hundreds of nodes. This introduces a significant management overhead. In general a system can easily be migrated to more powerful hardware with the same architecture to achieve better performance. However, using horizontal scaling allows the use of cheaper hardware. Michael et al. have shown that scale-out solutions are up to 4 times faster than equal priced scale-up solutions, even if enterprise hardware is used [154]. Furthermore, it has been demonstrated that clusters of off-the-shelf hardware enable very

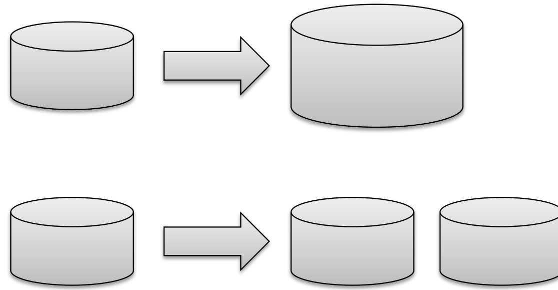


Figure 4.1.: Vertical (above) vs. Horizontal (below) Scaling

large scale clusters to be built at a fraction of the cost of high-end hardware [29, 37]. With the advent of multiprocessors and cloud computing, scaling out is the only solution in many large scale applications [194]. A further very important factor is that horizontal scaling can be done online, while the migration to a larger system usually has to be done offline.

The horizontal scalability of a database system depends on the scalability of the DBMS and the scalability of the database. The cluster database system architecture has a good horizontal scalability, as we identified above. The horizontal scalability of a database depends on its workload, it can be estimated by the speedup we defined in section 3.4 in equation 3.14. The theoretical maximum speedup can be estimated by calculating the upper limit:

$$\text{speedup}_{\text{max}} = \lim_{\#\text{nodes} \to \infty} \text{speedup} = \frac{1}{\text{serial}} \tag{4.1}$$

Obviously, the use of an infinite number of nodes is not efficient; we will give details on efficiency in section 4.2. The scalability of a system is a mere indicator of the capability of scaling. The scaling procedure itself introduces further challenges. For larger numbers of nodes in particular, the scaling has to be done automatically. In the following, we will give details of the procedure of scaling a CDBS and information on how to automatize the single steps.

## 4.1. Automatic CDBS Scaling

To scale a CDBS means adding a new backend node to the system. As indicated before, we will not give details on scaling the controller of the CDBS. Obviously, the larger the number of nodes in a system grows, the more nodes are necessary to achieve a noticeable increase of performance. Therefore, automatic routines are necessary to increase the manageability of a CDBS. Adding a new node involves several steps:

- Acquire a new backend system.
- Install and start the software on the backend.

- Integrate the backend into the CDBS.
- Migrate or replicate data on the backend.
- Synchronize the backend.

Acquiring a new backend means setting up a new or a spare cluster node and giving it access to the intercom. Obviously, this is a manual task in a traditional server room. If the system is run on a cloud platform, this step can be automated. Typical cloud platforms are, for example, Amazon's Elastic Compute Cloud [16] and Microsoft's Azure [122]. These provide APIs to acquire a new compute node automatically.

The complexity of installing the software can vary highly, based on the backend database system. Lightweight DBMSs, such as Apache Derby[1], HyperSQL[2], and H2[3] need little or no installation at all and can therefore simply be started on the node. Commercial systems often have complex installation procedures which do not allow an automatic approach [156]. A solution to automate the DBMS installation is virtualization, using preconfigured virtual machines eases the installation overhead. This is usually also used in cloud serving systems.

It is possible to load data to the node before integrating it into the middleware. However, then the database on the node has to be resynchronized at the point of integration. Therefore, it is easier first to add the node to the cluster and then ship the data. This way the middleware can also decide which data is placed on the node. To integrate the node into the cluster the middleware simply needs access to the DBMS on the node. Usually, management tasks such as bulk loading cannot be done via the SQL interface of the DBMS; these are also not standardized between different DBMSs. In order to automatize the integration into the cluster a slave program or a set of scripts is utilized. This usually has to be implemented for every brand of DBMS separately. When the middleware is aware of the node and it has access to the DBMS on the node, it can load the data on the node.

Depending on the size of the database the migration is a time consuming task. It consists of the export, the transmission phase, and the load phase, this is also known as ETL process [114]. In the export phase, the data is exported from the database on a running node of the system. The data is either stored in a file or directly transmitted. During the transmission phase the data is sent via the network to the new node. The new node either stores the data in a file first, or directly imports the data into the database. This is usually done using bulk loading techniques [220]. Data export and transmission obviously take linear time in relation to the data size. The import however has super linear costs because of the construction of indexes and the like. If the database engines are the same on the old and new node, the data can be simply transferred by copying the database on the file level. However, the database must be in a consistent state before and

[1]Apache Derby homepage - `http://db.apache.org/derby/` (last visited 2011-04-15)
[2]HyperSQL homepage - `http://hsqldb.org/` (last visited 2011-04-15)
[3]H2 Database Engine homepage - `http://www.h2database.com/html/main.html` (last visited 2011-04-15)

while transmission. This can be ensured by generating a snapshot of the database and logging all subsequent changes. In the CDBS architecture, the middleware can trigger a snapshot at any time and store subsequent changes or send them directly to the new node. In terms of scalability it is better to queue the updates on the node in order not to clog the controller. To automate the migration the ETL process has to be automated. This is feasible if the CDBS is homogeneous in terms of the software. If the backend DBMSs are very diverse, migration tools or scripts for all DBMSs are needed and mappings for the data representations.

When the snapshot is loaded into the new node, it has to be synchronized. Since the updates are stored on the middleware or queued on the node, they simply have to be processed. New updates will also be sent to the node. Obviously, it is important, that the updates are processed in the right order. When the node has an updated state, it can be fully operated in the cluster. If the updates are queued on the node, it has to inform the middleware after synchronization, otherwise the middleware automatically notices the state of the synchronization.

In this section we have shown how the CDBS scaling can be automatized. The automatic scaling not only allows the system to grow over time without excessive administrative overheads, but it also enables us to scale the system according to the load at any given point. To find an optimal scale the efficiency of the system has to be measured. In the following section, we will explain different forms of efficiency and show how they can be analyzed.

## 4.2. Efficiency of Distributed Systems

Although efficiency is a core principle in computer science, many large scale data processing systems have very low processing per node [30]. In order to increase the efficiency of a distributed system local as well as global optimizations should be taken into consideration. Local optimizations try to enhance the efficiency of a single system, while global optimizations improve the distributed system as a whole. In the CDBS architecture the efficiency of the backend DBMSs is interesting as well as the efficiency of the complete architecture. However, since we can assume that the backend DBMS is basically a black box, its efficiency can be estimated, but only improved on a limited scale. Therefore, we will not expand on this topic.

Efficiency does not only concern the processing per node, but also other metrics. Anderson and Tucek define the following types of efficiency for distributed, large scale data processing systems [25]:

**Compute efficiency** Compute efficiency is dependent on the amount of work a node can process in a certain time and how much work has to be processed. This is the most investigated form of efficiency. In theoretical computer sciences, this efficiency can for example be described by the big-O notation. For distributed database systems this is the throughput of queries per node. In a system that does not scale linearly, the efficiency obviously decreases with the number of nodes. For a CDBS

the compute efficiency can be measured by the speedup as defined in equation 3.14 in relation to the number of nodes:

$$\text{efficiency}_{\text{compute}} \sim \frac{\text{speedup}}{\#\text{nodes}} \tag{4.2}$$

Usually, the efficiency is a value equal or less than 1. In this special case, the efficiency may be greater than 1, since superlinear speedups are possible. However, this definition of efficiency is only valid for arbitrarily large problem sizes, i.e. arbitrarily heavy workloads. If a system has good compute efficiency according to the definition above, it is said to be *weak scaling*; if the system is also efficient for a fixed problem size, it is said to be *strong scaling* [22]. For an actual workload, the efficiency is therefore dependent on the utilization of the system. This relation can be described as follows:

$$\text{efficiency}_{\text{compute}} \sim \frac{\text{workload}}{\text{throughput}} \tag{4.3}$$

This equation implies that the system is not overloaded, i.e. the workload is not larger than the throughput. A system has an optimal compute efficiency if it is fully utilized while it has a maximum throughput per node.

**Storage efficiency** Storage efficiency is the ratio of used disk space to raw data size. In a database system this is disk usage versus the raw database size, without additional data structures. For distributed systems it is also interesting to measure the disk usage compared to a single node system:

$$\text{efficiency}_{\text{storage}} \sim \frac{\text{disk usage}_{\text{single node}}}{\text{disk usage}_{\text{cluster}}} \tag{4.4}$$

An approximation of the storage efficiency is the inverse of the degree of replication in the system, i.e. the number of replicas per table.

$$\text{efficiency}_{\text{storage}} \sim \frac{1}{\text{degree of replication}} \tag{4.5}$$

If the degree of replication varies highly between different tables or database partitions, the average can deviate greatly from the actual disk usage. Because of the fault-tolerance a distributed system usually has less then 50% storage efficiency.

**I/O efficiency** I/O efficiency is the goodput of a network or disk in comparison to the theoretical maximum throughput of the device. The goodput is defined as the amount of useful bits transferred per second. The goodput is dependent on the protocol overhead as well as other factors such as replication. Replicated data increases the network traffic and therefore decreases the I/O efficiency, since updates

have to be replicated, i.e. sent via the network to all replicas. However, the use of multicast techniques can greatly improve the efficiency of a network [56]. In the best case, the multicast reduces the amount of traffic in a replicated CDBS to that of simple client server system. Since the CDBS architecture is optimized to minimize network traffic resulting from query processing, network efficiency in general is low. However, network efficiency is more important with scaling. Usually, a network can be fully utilized in bulk data transfers. If network bandwidths differ between nodes, protocols such as the Fast Send Protocol (FSP) can be utilized to speed up data transfers [186]. Obviously, the disk I/O efficiency for query processing depends mainly on the backend DBMS.

**Memory efficiency** Memory efficiency determines the degree of memory waste compared to an optimal representation. This is an important factor; unlike CPU or network contention, memory exhaustion usually causes systems to fail or at least abort tasks. For DBMSs this means that memory intensive queries may be aborted. In the CDBS architecture, the memory efficiency of the backends cannot be directly influenced. However, advanced allocation schemes enable the memory footprint of a query during processing to be reduced. This will be further discussed in chapter 7.1. The memory efficiency of the middleware is dependent on the data structures for node management and query processing.

**Programmer efficiency** Programmer efficiency measures the speed at which programmers can implement a given task with different systems. In DBMSs this is the speed in which users can express queries. Many distributed database systems use non transparent partitioning and distribution of the data, which is also known as *sharding*. Furthermore, the current non-SQL movement recommends using concurrent programming languages like Erlang [27]. However, since distributed computing is highly complex [125], transparent access to the distributed data increases the programmer efficiency. The SQL language features a declarative model that also increases the programmer efficiency in comparison to imperative models [148].

**Management efficiency** The management of a system is efficient if the number of people needed to manage the system is minimal. It can also be described by the time needed to manage a system. This efficiency can be increased by automating management tasks, reducing system complexity and using known, well documented technology. Obviously, the automation of scaling and data movement is an important increase in management efficiency. Even if tasks are automated, they usually have to be triggered by administrators. The decision if a certain management task is executed is often also difficult and time critical. For this reason self-managing systems were introduced, especially in the database domain they are often called autonomic systems. We will give details on autonomic systems and how to introduce autonomic computing to CDBSs in section 4.4.

**Energy efficiency** The amount of energy used by a system compared to a system with

minimal energy requirements specifies its energy efficiency. Energy efficiency is related to other forms of efficiencies, since a system with bad compute efficiency will usually use a lot of energy. The same is true for storage and I/O efficiency. In distributed systems, the scale of a system, i.e. the number of nodes, has great influence on its utilization and therefore the energy efficiency of a system. Obviously, the energy efficiency of a system is hard to quantify. Therefore, the efficiency improvement of a system is usually quoted:

$$\text{improvement}_{\text{efficiency}} \sim \frac{\text{energy usage}_{\text{old}}}{\text{energy usage}_{\text{new}}} \quad (4.6)$$

Obviously, this metric does not allow valid comparisons of different systems. Apart from that, there are many factors that can be considered. Besides the energy consumption of a running system, the energy consumption for networking, cooling, hardware production, and even lightening can be taken into account [32]. Hence, standardized metrics differ in their factors and measurement requirements [181]. For DBMSs the TPC Energy standard specifies the Watts per transactions metric [230]; this metric enables the energy efficiency of DBMSs to be compared based on the TPC benchmarks (see section 17). However, the load of the system under test is not considered, it is assumed that the system is constantly running at its maximum performance. As we stated above, most systems seldom run at their maximum performance; valid comparisons would need to measure the energy consumption at a specified throughput. If the throughput varies over time, the efficiency of a static system will vary, therefore dynamic techniques are necessary to increase it. We will give details of ways to increase the energy efficiency from a software architectural point of view below.

**Cost efficiency** The cost efficiency of a system is obviously also related to the other forms of efficiencies. More compute efficient systems require less hardware, which reduces costs of acquisition. The same is true for storage, I/O, and memory efficient systems. Programmer and management efficiency reduce the costs of labor and energy efficiency reduces the cost of energy consumption, which is apparently the main driver for energy efficiency. Apart from that, the use of commodity hardware can significantly reduce the cost of a system and therefore increase the cost efficiency [175]. A typical metric related to the cost efficiency are the TPC price/performance metrics, these measure the cost of a system per transactions per second. However, they only consider the acquisition and maintenance costs.

The CDBS architecture aims to increase all forms of efficiency. Apart from programmer and memory efficiency, all forms of efficiency benefit from scaling the system to the right size. Further gains can be achieved by automatically adapting the system to the current load. In the next section, we will discuss how dynamic scaling helps to increase the energy efficiency of a system; after that we will present the principle of feedback control loops. These allow an autonomous optimization of arbitrary system properties.

## 4.3. Energy Efficiency of Scaling

As stated above, distributed systems can be optimized locally or globally. Examples of local optimizations for energy efficiency are dynamic voltage scaling [226] and switching off devices such as hard drives. An exhaustive survey can be found in [221]. Global optimizations are based on global decisions such as switching off complete nodes or prioritization of services [152]. Naturally, the combination of both options brings the best results. However, recent research has shown that global optimizations result in a higher benefit than local ones [192]. Obviously, the number of processing nodes in a CDBS is an important efficiency factor. Too few nodes will not be able to process the given workload; an over scaled system will contain many nodes that are idle or only used rarely. It is common practice to use a system scale, that has enough resources to process peak workloads easily. Even if the peaks occur very frequently, most of the time the system is underloaded. An example of this behavior can be seen in figure 4.2: it shows the number of requests per second at the Wikimedia clusters, host of the Wikipedia website.

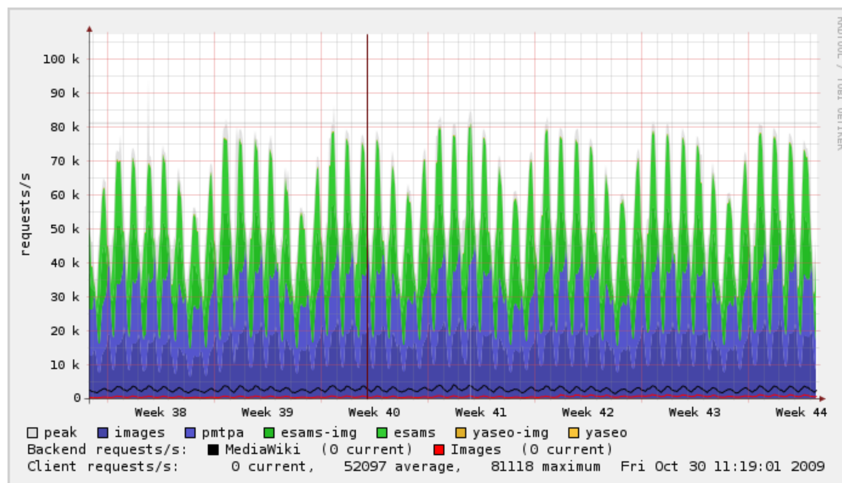



Figure 4.2.: Requests per Second at the Wikimedia clusters in October 2009 in Europe (green) and the USA (blue) (image source: `http://en.wikipedia.org/wiki/Most_viewed_article`).

The workload of the Wikimedia clusters is variable and the average load is only about 65% of the maximum load. So scaling the system according to the active workload could reduce the number of nodes on average by 35%. Furthermore, server systems usually have a very low utilization, often as low as 6% [134]. Adequate scaling can increase the utilization dramatically. Contrary to common belief it is possible to run many systems efficiently on 100% CPU utilization [140]. It has to be pointed out, however, that a constant utilization of 100% is usually not desirable in a dynamic environment if the danger of overloading a system is high. Besides the workload characteristics, the adequate utilization depends on the duration of the scaling process and the temporary performance

loss due to the scaling. If the scaling process is very costly in terms of resources, the general utilization has to be lower than if the scaling process is cheap. Obviously, the cost of the scaling process reduces the overall efficiency of the system. Because of the many variables that have to be considered in the scaling process, autonomous techniques are needed in a dynamic environment.

## 4.4. Autonomic Computing

There are different approaches to adapting a system to dynamic environments, they can be classified in three categories: *static*, *proactive* and *reactive*. A *static* approach defines a single state of configuration. This state is usually chosen to be optimal on average and adequate in all situations. For the scale of a DBS this means that it is large enough to process incoming workload peaks and as small as possible to maximize the efficiency of the system. *Proactive* and *reactive* systems change the state of the system to adapt it to the environment. While *proactive* approaches anticipate the changes in the environment and prepare the system for the future environment, *reactive* approaches adapt a system if certain changes in the environment are detected. Proactive strategies are possible if the changes in the environment follow known rules or if external knowledge enables the future changes to be foreseen. Reactive procedures are especially useful in dynamic environments with infrequent, unexpected changes. In general, reactive solutions are easier to implement and are computationally less expensive. However, reactive strategies have - depending on the managed entity - a considerable delay, resulting in efficiency or even performance decrease. Proactive strategies, on the other hand, can increase efficiency if the anticipation of the future workload is reasonably accurate. If it is not, they either fall back to the reactive case or tend to over-provision.

If a system is capable of adapting itself proactively or reactively, it is called self-managing. IBM coined the term *autonomic computing* for technologies of self-management for distributed systems [137]. Self-management consists of the four concepts: self-configuration, self-optimization, self-healing and self-protection. Self-configuration refers to the ability to install, configure and integrate a system or component autonomously. The autonomous and continuous improvement of performance and efficiency is called self-optimization. Autonomic detection and repair of problems and attacks is subsumed under the terms self-healing and self-protection, respectively. In the following, we will concentrate on self-optimization, but will also consider the other concepts.

To enable a system to react or proact autonomously a control loop can be used. These usually have a form like the MAPE model [3] or the OPR model [223]. The *online feedback control loop* described by MAPE consists of four phases, *monitor*, *analyze*, *plan* and *execute*, while the OPR loop consists of three phases: observation, prediction and reaction, in which the analyze and plan phase of MAPE is merged into the prediction phase. In the following, we will further outline the better known MAPE loop. The phases in MAPE can be seen in figure 4.3. In the MAPE model a single loop concentrates on a single managed element or resource. The element's state can be measured via

sensors and manipulated via effectors. The interface of a set of sensors and effectors is called touchpoint. Using various touchpoints an *autonomic manager* can automate some management tasks concerning the managed element by providing the four functions for the phases of the control loop. The autonomic manager uses knowledge that can be accessed by all of its functions. This knowledge can include metrics, thresholds, logs and other information. The knowledge is either provided externally or gathered by the manager itself. The managers functions, i.e. the phases of the control loop, have the following functionalities:

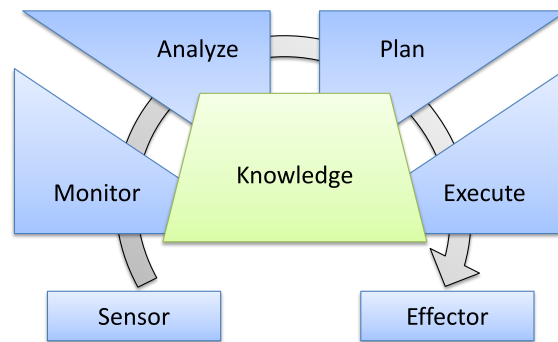



Figure 4.3.: The MAPE Loop

**Monitor** Using the monitoring function the system gathers information on its state. In order to get the information the system uses sensors that continuously or periodically measure the state of the managed element. The information of the sensors is collected, aggregated and filtered. The resulting information is also referred to as symptoms. This information can also be incorporated into the knowledge base of the manager. The monitoring can be a continuous process that is not interrupted by the other phases.

**Analyze** The analyze function checks the symptoms that are provided by the monitoring function. It determines if all requirements, such as policies or thresholds are met. These requirements are also called constraints. In order to predict the trend of the system behavior, the analyze function can employ time-series analysis and the like. If it decides that a constraint is violated, it can send a change request to the plan function, which includes the state changes needed.

**Plan** If the analyze function has found a dysfunction, the plan function develops a change plan, i.e. a program or workflow that includes the desired changes of the managed element in order to obtain an acceptable system state. The system is in an acceptable state if all or enough policies and thresholds are met.

**Execute** The execute function schedules the change plan and executes the required actions that are specified in the change plan. It uses the effectors of the managed element. The execute function might also update the knowledge base.

Apart from the autonomic management, the MAPE model includes a manual management interface, this allows a user or external program to perform the management tasks by bypassing the autonomic manager. The MAPE model enables autonomic management to be introduced to managed resources that provide interfaces that act as touchpoints and knowledge about the management task. As a part of this thesis the MAPE middleware Scalileo was implemented for managing and, in particular, scaling distributed systems.

# 5. Scalileo

The Scalileo framework is a middleware system that is based on the MAPE model. The main focus of its design was to scale and migrate distributed Java applications autonomously. It is highly configurable and easy to extend. In the following chapter we will describe the architecture of the system and use it to scale a distributed web server application. Goal of the prototype is to give a proof of concept that the framework is able to autonomously manage a distributed application. Since we aim for efficiency, we use the scaling framework to increase the energy efficiency of the distributed web server. As we argued in section 4.2, energy efficiency is related to multiple forms of efficiency; furthermore, increases in energy efficiency are relatively easy to measure. Therefore, we use energy efficiency as a showcase for other forms of efficiency.

## 5.1. Scalileo's Architecture

Most distributed applications feature several worker nodes that carry out computationally expensive tasks, as well as a central component which is responsible for organizing and controlling the entire system (see section 3.5). In these systems not all of the nodes have the same function, as the central component has different tasks to the worker nodes. Therefore, the Scalileo framework implements a multi-tier architecture. It uses a central component, called master node, which is responsible for organizing a set of workers running on different physical nodes. The workers are controlled by the master and process its commands. Furthermore, they perform benchmarks to monitor the status of their corresponding node. An overview of the architecture of Scalileo and its relation to the distributed application can be seen in figure 5.1. This architecture also reflects the CDBS architecture. As far as the MAPE model is concerned, the master implements the autonomic manager, while the workers act as touchpoints.

### 5.1.1. Workers

A Scalileo worker is a process, running on a specific physical node that is host to a process of the distributed application or may be host in the future. Its main task is to spawn or shut down processes of the distributed application in order to increase or decrease its scale. The distributed application can supply a set of files and a set of commands to be executed on the host machine for this task. These will then be transferred by the Scalileo system. By doing so, the application can transfer an executable program as well as initial data needed by the distributed process that will be started. Thus, the remaining task of the distributed process is to integrate itself in the distributed application, e.g. by

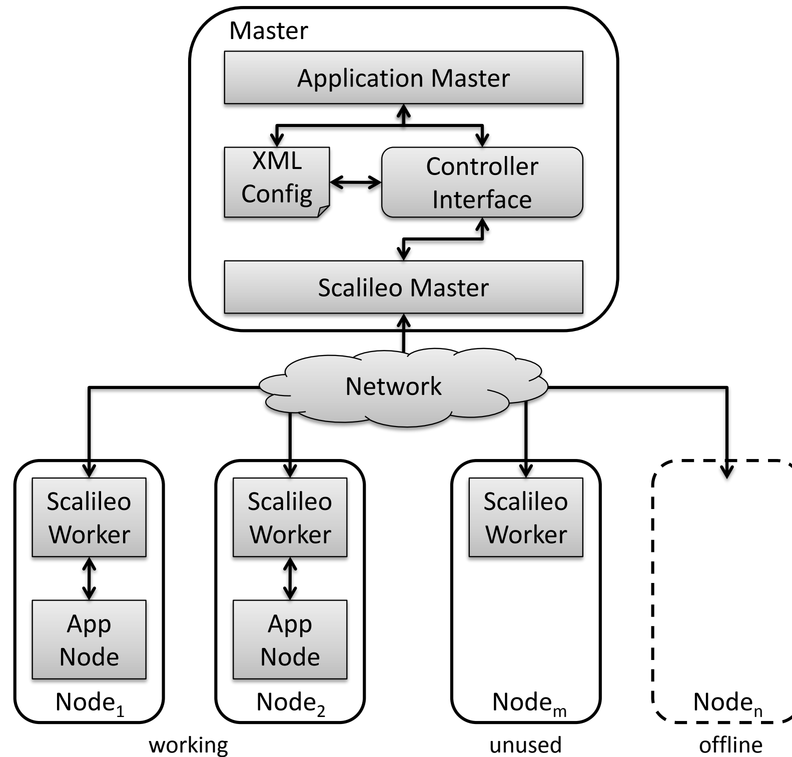



Figure 5.1.: Overview of the Scalileo Architecture

registering itself at a superior instance. If the node is not needed any more the worker can shut down the spawned process by executing a command that is provided by the application. If specified, it then transfers the data back from that node.

The second task of a worker is to run benchmarks on its system. These can be measurements of system-wide performance parameters like load, CPU utilization or free space but also special performance parameters provided by the distributed process that runs on the machine at the time. These data are acquired in specified intervals and are sent to the master node for evaluation.

Furthermore, the workers can perform any management tasks that is commanded by the master. For each task an interface to the managed element or a set of scripts or executables has to be provided.

### 5.1.2. Master

A worker deals with the tasks that are processed on a single node. In contrast, the duties of the master node are the organization and management of the worker nodes and keeping track of the global state regarding the overall performance and efficiency of the distributed application. The main tasks of the master are:

**Initializing workers** First, the master starts a worker on every node that may be used to run processes of the distributed application. It will log on to each node, transfer the worker's executable file and start the worker process.

**Collecting benchmark results** The master node collects the benchmark results that are measured by the workers. The workers send the benchmark results back to the master node and the master receives these results using a special listening thread. If the master node also has to be managed, it will also have a worker running.

**Tracking system state** As the central instance in Scalileo, the master node is the only one aware of the benchmark data of all workers in the system. Using this information, the master will reduce the values for each benchmark type to a single value. For example the free space of every single node in a distributed storage system is accumulated to a single value describing the free space still available on all active nodes in the system. It is also possible to aggregate the reduced values of different benchmarks into a single combined value.

**Maintain specifications** The master node constantly monitors the benchmark results and compares them to a set of preset constraints that define upper and lower boundaries between which these values should lie. If a certain value exceeds such a boundary for a specified time, the system needs to be adapted. If this is the case the master node informs the distributed application that an adaption is necessary. In the case of a scaling, the master passes a list of nodes to the application that can be added to the system or removed from it. This list is ordered according to the suitability of the nodes: the most promising nodes are at the top of the list. If a storage system, for example, runs out of free space the nodes with the most available space are ranked first.

**Adapt the system** The master plans the adaption of the system and informs the managed application of its plan. The application can either conform with the plan, alter the plan or reject the plan completely. If the application conforms with the plan or provides an adapted plan, the master will execute it. For scaling the plan is a scale up or scale down request. If the distributed application receives a scaling request from the master it decides if it complies with the request; it then picks a node from the given list. As the easiest option, the application can take the first element on the list. It is most promising to solve the problem in correspondence to the benchmark data. However, it can also independently choose a node for scaling or even decide not to scale at all. If a node is chosen for scaling, the application provides a command to the master node either to spawn a node or shut it down. Additionally, a set of files can be defined that is transferred to the node or back from it. Then the master node first spawns a worker process on that node (if not already running) and secondly executes the command given with the help of the worker.

As the master node has to communicate with the distributed application in order to perform the adaption, an interface is necessary. In Scalileo this is achieved by implementing the *Controller* interface. It contains all necessary methods to cooperate with the master node and to provide the required data. We will describe the different interface methods in detail below. Currently, the interface reflects Scalileo's purpose for scaling:

**constraintViolated** This method informs the controller that a constraint has been violated. This might be a policy that is not met or a threshold that is exceeded. Such an event might not necessarily lead to an adaption or scaling, as the violation could be temporary and therefore not long enough to initiate an action, but it gives the application the chance to react to certain changes of the system's state.

**beforeScale** This method is used by the master node if it determined that scaling is necessary and the violated constraint is passed to the application. It allows the application to prepare for the scaling process.

**chooseNode** In order to choose the correct node for scaling, an ordered list of nodes is passed to the application by the *chooseNode* method. The application returns the node that should be used for scaling or *null* if no scaling is desired at this time. This method gives the application the possibility to alter the scaling process.

**getNodeSetup** If a node was chosen for scaling, the master node will ask for the setup of the node through the *getNodeSetup* method. The setup contains the command to be executed, the files to be transferred to the node as well as the target directory, where the files should be copied to and where the command will be executed. The executed command has to terminate after it has started the desired process and return 0 if this was successful and a value greater than 0 if a problem occurred. This method allows the manager to access the knowledge provided by the application.

**getNodeShutdown** In the case of a scale-down event, the manager will ask for a shut down command using this method. The application provides the command for shutting the application down. It can optionally specify a path to a file or directory on the node, which will be zipped and transferred back to the master.

**afterScale** After the scaling process the application is informed of the result. With this information the application can determine if the scaling was successful or if an error occurred. The method returns an error code if an error occurred before the command could be started. If the command could be executed the return value of this command is returned instead.

There are further methods for informing the controller of certain events, like receiving benchmark results, handling errors such as hardware failure, etc. However, since many of these methods are not necessary for every application, Scalileo offers an *AbstractController* class, which already implements most of these methods with default behavior. This reduces the programming effort, for a basic setup a programmer only needs to implement the two obligatory methods for retrieving the node setup and shutdown commands.

### 5.1.3. Parameterized Components

A certain type of component exists for every task in Scalileo that either specifies how to do this task or holds the necessary data for it. To make Scalileo extensible and to achieve a high adaptability for most scenarios and applications, the components have a common structure so that the complete configuration of Scalileo can be defined within an XML file. Scalileo uses the Java Reflection API, which enables objects to be constructed at runtime by using metadata about the object's class. The metadata stores information such as the name of the methods contained, the name of the class, the name of parent classes, and/or what the compound statement is supposed to do. Using this information, an object can be created by reflecting upon the given class name and determine if that class exists, what kind of operations it supports, which interfaces it implements and what parent classes it has. This gives Scalileo the ability to specify its components in XML and to construct the corresponding Java object at runtime with this information. Therefore, it is not necessary to hard-code any predefined components. It is sufficient to specify only certain interfaces which define the methods that the implementation of a certain component must provide.

In the configuration file every Scalileo component is specified with a certain tag, depending on the type of the component. The tag requires two attributes: an ID for referencing this component and a fully qualified Java class name that specifies the Java class which will be implementing this component. Consider the following example that implements the login process to nodes via the Secure Shell protocol (SSH):

Listing 5.1: XML Snipet for the SSH Login

```
<login-method id="sshLogin" class="scalileo.login.SSHLogin" />
```

The chosen ID is "sshLogin" and the implementing class is scalileo.login.SSHLogin. To instantiate this class via the Java Reflection API all classes must follow a common architecture. All classes that implement a Scalileo component must be derived from a specified abstract class, depending on the component's type. Common to all these classes is that the class must have a public constructor that takes two arguments: an ID of type java.lang.String and a set of parameters of type java.util.Map<String,Object>. Every parameter in this map is identified with a key and has a java.lang.Object as a corresponding value. Additionally, a component must provide the method *hasValidParameters*, which is invoked after constructing the object to determine if all necessary parameters are set. In this method the programmer must ensure that the parameters set in the constructor are complete and valid. If they are not valid, the component will not be instantiated.

Listing 5.2: Interface for Components

```
public <constructor>(String id, Map<String,Object> parameters) {}
public abstract boolean hasValidParameters();
```

The parameters used in the constructor are specified in the XML file when a certain instance of this component is defined. Following the example above, an instance of a login component is defined in the XML file at the specification of every node. For the

SSH login example two parameters for user name and password must be passed to the login component. In the node definition in the XML file a login-method must be defined in the <login−with> element, so that Scalileo knows how to access the specified node. As login type the ID of the login-method must be given. Additionally the <login−with> tag can contain several <parameter> tags that define the parameters for this component. A parameter is identified with a key and must specify the type of the parameter which must be a fully qualified Java class name. This class must provide a public constructor that takes a String as an argument, which will become the content of the value attribute. Thus, every class that provides such a constructor can be used as a parameter object. The Java framework already provides many possibilities for this as for example all primitive types like String, Integer, Float, Boolean or more complex objects like java.util.Date. A node specification in the XML file could therefore look like this:

Listing 5.3: Specification for a Node Login

```
<node name="node-01" address="node1.example.com">
  <login-with type="sshLogin">
    <parameter key="username"
      type="java.lang.String" value="alice" />
    <parameter key="password"
      type="java.lang.String" value="secret" />
  </login-with>
  ...
</node>
```

When Scalileo parses the node definition in the XML configuration file it will first instantiate all defined parameter objects and save them in a parameter map object. Then the specification of the given login type *sshLogin* is looked up and the corresponding class scalileo.login.SSHLogin will be instantiated with the components *ID* and the parameter *map*. When Scalileo later needs to log on to that node this login-component will be used. All other components are defined following the same principle.

In the following, the different Scalileo components are described. Figure 5.2 depicts the internal control cycle of Scalileo and its relation to the MAPE model. It shows the most important components with their relationship and interaction.

### 5.1.4. Benchmarks

The components for measuring performance and other properties are called benchmarks, these perform the tasks of the sensors in the MAPE model. All benchmarks must extend the abstract class scalileo.benchmark.Benchmark, which requires - in addition to the requirements mentioned before - a method *run* that returns a Double value as result of the benchmark.

Listing 5.4: Interface for Benchmarks

```
public abstract double run() throws BenchmarkException;
```

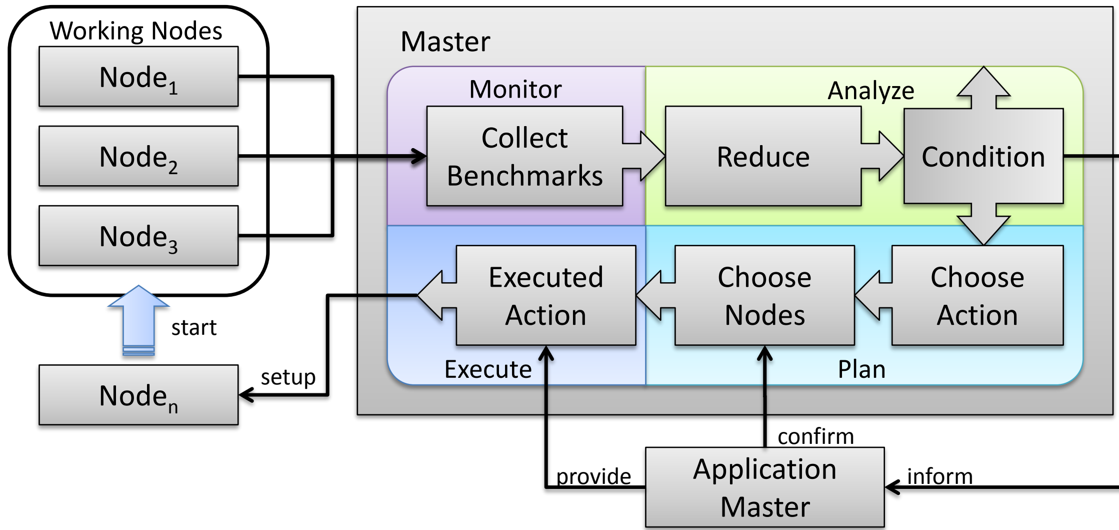



Figure 5.2.: Scalileo's Feedback Control Loop

The Scalileo Benchmark class provides functionality to repeat a benchmark at given time intervals. This can be achieved by specifying an Integer parameter called *interval* which defines the interval length in milliseconds by which the benchmark will be repeated. In the XML configuration file an example benchmark definition could look like this:

Listing 5.5: Specification of two Benchmarks

```
<benchmarks>
  <benchmark id="ExampleBenchmark"
     class="example.package.ExampleBenchmark" />
  <benchmark id="OtherBenchmark"
     class="example.package.OtherBenchmark" />
</benchmarks>

<node ...>
   ...
  <use-benchmark type="ExampleBenchmark">
    <parameter key="interval"
       type="java.lang.Integer" value="10000"/>
  </use-benchmark>
  <use-benchmark type="OtherBenchmark">
   ...
</node>
```

When a worker process is spawned on a node by the master, the benchmarks assigned to this node are transmitted and the worker will ensure that the benchmarks are executed

in the given interval. If no interval is specified, the benchmark will only be run once at the start of the worker process. This is used for benchmarks that measure static parameters like CPU frequency or other primarily hardware related parameters.

Scalileo comes with a set of predefined benchmark components like a PingBenchmark class, measuring the round-trip time of a ping packet to a certain host; and a benchmark for measuring the available disk space on a certain file system. All predefined benchmark classes are located in the scalileo .benchmark package. It is also possible to use already available measurements, for example from monitoring systems such as Ganglia [151]. They only have to be wrapped by a implementation of the *Benchmark* interface.

### 5.1.5. Reduction

When the master node receives a benchmark result from a worker it will update all constraints affected by that benchmark. Before this can be done, the benchmark results must be merged into a single value. To do this, Scalileo uses reduction components that will reduce a set of values into one aggregated value. Reduction components must be derived from the abstract class scalileo .reduction.Reduction which specifies one method for performing the reduction. This method takes a collection of Double values and will return the reduced value of this collection:

Listing 5.6: Interface for Reduction

```
public abstract double reduce(Collection<Double> results);
```

The Scalileo package includes a number of reduction components, covering the most common reduction functions. Among others these are components for reducing a given list of values to their maximum, minimum, sum, average or median value. Apart from simple aggregation, the reduction component also enables filtering to be implemented.

### 5.1.6. Conditions

The next components in the Scalileo framework are so called conditions, which the reduced values are compared to by the master node in order to determine if the system's performance is still in the desired state. Those conditional components must be derived from the abstract class scalileo .condition.Condition and therefore implement a *check* method that takes a reduced Double value as its argument and return a Boolean value, which indicates if the condition is met or not.

Listing 5.7: Interface for Condition

```
public abstract boolean check(double value);
```

Three conditional components are predefined in Scalileo, one to test if a value is equal to another value (EqualCondition), if it is smaller than an upper bound (MaxCondition), or if it is greater than a lower bound (MinCondition). It is also possible to specify several conditions which are connected with a logical AND conjunction, so it is for example possible to specify an interval in which a value must be by combining a MinCondition and a MaxCondition.

### 5.1.7. Constraints

The three components mentioned earlier, benchmarks, reductions and conditions, are combined into so-called Constraint components. A constraint defines a boundary for the values of a certain benchmark. If the values are not within this boundary, the constraint is violated. Therefore, a constraint specifies one benchmark component whose values are reduced to a single value with a specified reduction component and defines one or several conditions which must be met by the reduced value. The constraint does not only observe the last known value, but tracks the values over a specified period of time. Only when the constraint is violated over this period is a specified adaption performed. This avoids, for example, an expensive scale operation being executed due to a single short-term peak or a temporary slowdown of a node. The results of a reduction are tracked over a time period and are averaged over this period. Only when this average exceeds the defined boundaries is an adaption operation suggested by the Scalileo framework. Like all components of Scalileo constraints are defined in the configuration XML file. An example of a constraint definition can be seen below.

Listing 5.8: Specification of a Constraint

```
<constraints>
  <constraint id="SomeID" historyDelay="20000">
    <use-benchmark type="ExampleBenchmark" />
    <reduce-by type="MaxReduction" />
    <check-condition type="MaxCondition">
      <parameter key="max" value="45"
            type="java.lang.Double" />
    </check-condition>
    <scale-action action="addNode">
      <choose-by benchmark="ExampleBenchmark"
            better="lower" weight="1" />
    </scale-action>
  </constraint>
  ...
</constraints>
```

The constraint shown above uses the ExampleBenchmark to measure the state of the system. The benchmark could be, for example, the CPU utilization or the network delay. If the maximum measurement of ExampleBenchmark on all nodes is above 45 in the last 200 seconds an addNode scale action is issued. To choose a suitable node the ExampleBenchmark is used once more.

Two types of scale actions (addNode and removeNode) can be used in a constraint, causing the adding or the removal of a node. It is also possible to define both types in one constraint in order to achieve a replacement of a node. For each action the controller is given a list of nodes ordered by their suitability for being removed or added. This order is created by the master node by comparing the nodes with the help of *choice* methods. A

choice method defines a benchmark component whose results are used in order to compare the nodes in an ascending or descending order. This order depends on whether higher values or lower values are preferable. Nodes can be compared by means of several choice methods, so that the ranking is not based on the values of a single benchmark, but on a set of benchmarks.

The order algorithm is implemented following the Java Comparator interface, which in this case takes two nodes (n1 and n2). It returns a negative integer, zero, or a positive integer as the first node is more suitable, equally suitable or less suitable than the second node. To compare the suitability of two nodes a point system is used. The nodes will be compared for every choice method defined. At each of these comparisons every node is given a certain number of points. If the current choice method states that higher values are better, the points for node n1 result from the benchmark value $b_{n1}$ of n1 divided by the benchmark value $b_{n2}$ of n2. The number of points assigned to n2 is the reciprocal value of this fraction. So if n1 has a higher benchmark value, it will get more points than n2 and thus be evaluated as more suitable in regards to this choice method. The points for each choice method are summed up to a total number of points $p_{n1}$ and $p_{n2}$. The node which achieves a higher point number at the end will be ranked higher on the list. Additionally, every choice method can be weighted to increase the influence of a certain benchmark on the order of the nodes. The points of every round are multiplied with the weight $w_c$ of the choice method before they are added to the total number of points, where $c$ is a choice method and $C$ is the set of all choice methods. The calculation of points corresponds to the equations 5.1 and 5.2.

$$p_{n1} = \sum_{c \in C} \begin{cases} \frac{b_{n1}}{b_{n2}} \cdot w_c, & \text{if higher values are better} \\ \frac{b_{n2}}{b_{n1}} \cdot w_c, & \text{if lower values are better} \end{cases} \tag{5.1}$$

$$p_{n2} = \sum_{c \in C} \begin{cases} \frac{b_{n2}}{b_{n1}} \cdot w_c, & \text{if higher values are better} \\ \frac{b_{n1}}{b_{n2}} \cdot w_c, & \text{if lower values are better} \end{cases} \tag{5.2}$$

### 5.1.8. Login Methods

To enable Scalileo to access nodes running under arbitrary operation systems and environments, the login process is encapsulated in separate login components. When the master node has to spawn a worker on a certain node it needs access to the node first. It further needs to have the ability to run commands on this node and transfer files to it. The login component's task is to provide this functionality to the master node. It must be derived from the abstract Java class scalileo.login.Login. The derived class must implement one method for transferring data and running a program on the target node.

Listing 5.9: Interface for Login

```
public abstract void runProgram(AppSetup setup);
```

The AppSetup object is the argument of the *runProgram* method. It contains a command string that is executed, a path string to a target directory in which the command is

executed and a file object that contains a link to the file that is transferred to the target directory before running the command. To transfer several files at once, the files must be zipped or compressed to a single file which can then be transferred via the login method. The command can then extract the file and run the command. Scalileo comes with one login component implemented which is able to log in via Secure Shell (SSHv2) on the node. SSH is available for nearly every platform and is considered to be secure. Thus, Scalileo can cover a wide variety of node types.

Using the implementations provided, a variety of distributed applications can be augmented with autonomic scaling. As a first show case a distributed web application was implemented, we will discuss the implementation and the performance results below.

## 5.2. Web Server Application

In order to test Scalileo we chose a distributed web server. Similar to CDBSs, web servers are exposed to dynamic workloads and are often implemented as distributed or replicated applications. In contrast to a CDBS, each node in a distributed web server is a completely independent application. There is no need for communication between the nodes, since every node can serve incoming requests independently. This enables us to add and remove nodes from the system very easily. In our example the data set is static, if requests would change the data set, a common database is needed. This could, for example, be implemented by a CDBS, as presented in the next chapter 6. The system consists of a central dispatcher server that distributes incoming requests to a set of up to 4 worker machines, which handle the actual request. The dispatcher uses round-robin scheduling, which is also used in DNS servers for load balancing. The redirection from the dispatcher to a worker machine is done by a HTTP 302 redirect (Found), so that the client will generate a new request to the corresponding worker. A overview of the test setup can be seen in figure 5.3.

The workload produced by the clients in this test was based on real-world system loads. We used workload traces of the web based E-learning management system Stud.IP at the University of Passau, which handles requests from a total of 15,000 users, consisting of different user groups – students, teachers and administrators. Like most web-based systems, Stud.IP shows a significant variation over a day, a week and even a year. The load is high during working time on days in during the lecture period, whereas at night time, weekends or semester break the load is comparatively low. We will give a more detailed description of the system and the workload in chapter 17.3, a sample of the workload can bee seen in figure 5.4.

In our test we set up the clients so that the workload during the test simulates the first day of the lecture period of the Stud.IP system. We differentiate between two types of requests: dynamic and static websites. The response returned to a dynamic client request from the web servers was a web page containing an image, which was resized for every request by the web servers. The computational effort was therefore mainly dependent on this image resizing. For static requests we chose to return a simple HTML document.

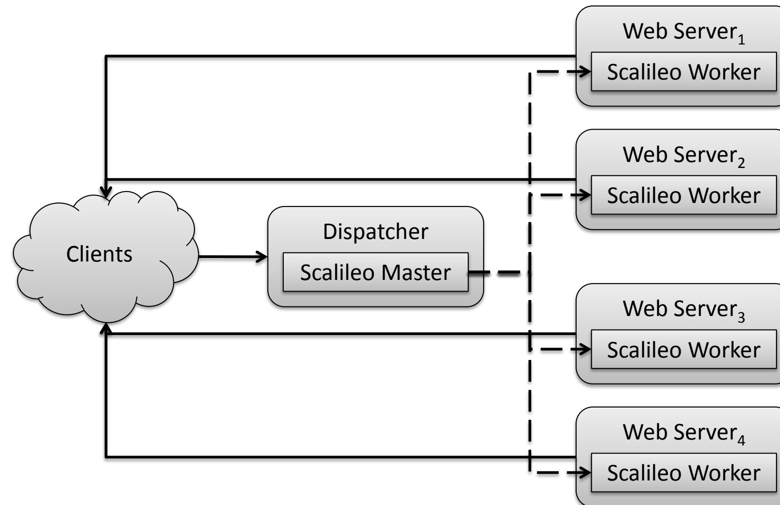



Figure 5.3.: Scalileo Web Application Setup

The workload was chosen in a way, so that the system could easily handle the requests at peak times when using all four available web servers. Accordingly, at times with reduced load, it is possible to reduce the number of worker servers while still being able to handle all requests. Aggregated in 10 minute steps, the original workload has a peak of about 80 requests per second and is on average 25 request per second. To reduce testing time and increase the workload, we sped the replay of the trace up by a factor of 48 and used only every 20th request, resulting in a peak load of 192 requests per second and an average load of 60 requests per second. The ratio between static and dynamic accesses varies between 0.2 and 66.4 and the median is 0.3. Hence, on average every fourth requested web page is dynamic. As the average load is only a third of the peak load, it is likely that reducing the number of servers in times of lower load can lead to a drastic reduction in energy consumption. It has to be pointed out that the first day of the lecture period has the highest workload in the whole year. Accordingly, for the complete year even higher energy savings will be possible.

Our tests were conducted on 6 workstation PCs with Intel Pentium D 3 GHz Dual Core Processors, 3GB RAM and 100 Mbit/s Fast Ethernet. The OS is Ubuntu Linux 8.04.2, Kernel 2.6.24-23 and the used Java version is 1.6.0_16. The workstations have a power consumption of 91 Watts when idle, 200 Watts during boot phase and 2 Watts when they are switched off. All power measurements were conducted using a PCE-PA 6000 power analyzer.

We used the Scalileo framework to implement a distributed web server. The Scalileo master was running on the same machine as the dispatcher server. On every running worker machine a Scalileo worker node measured the processor load on the machine and reported back to the master as benchmark results. If the overall load on the system was beyond a threshold, Scalileo started a further web server worker on a new machine to

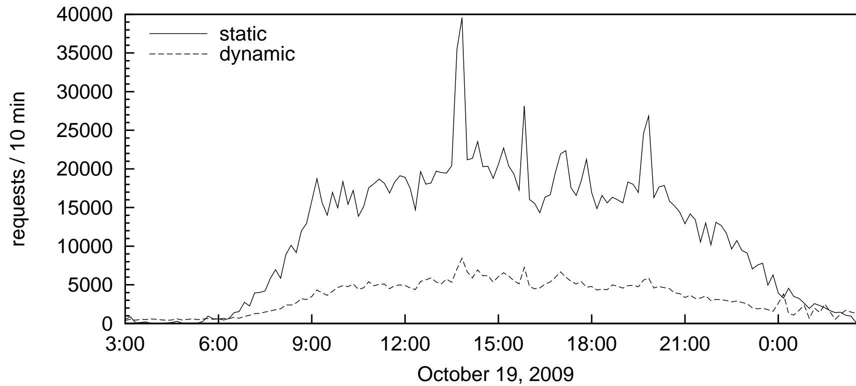



Figure 5.4.: HTTP Workload Trace of Stud.IP at University of Passau for the First Day of the Winter Term 2009

increase the system's performance. To save energy the Scalileo system shut down unused machines and woke them up via Wake-on-LAN when they were needed again.

Since booting and halting of the machines consumes energy without any value for the distributed web server, it had to be assured that scaling did not take place on small and temporary load changes but only when necessary. The constraints for the application were determined iteratively and the final values were as follows: if the average CPU usage over all active nodes was higher than 45% in two thirds of all benchmark samples over 20 seconds in the simulation, a new node would be spawned. This corresponds to a period of 16 minutes in original speed. At the lower boundary the CPU usage had to be lower than 20% over 35 seconds before a node was shut down. This corresponds to 28 minutes in real time. In a real world setup, both thresholds could be set higher, but the high simulation speed enforced these parameters in order to give the system enough time to spawn a new node.

We focused on two goals in the test: on the one hand, the reduction of the necessary energy to operate the system and, on the other hand, a stable response time for the requests. It is clear that both goals had to be met, as energy saving should not happen at the cost of increased response times.

In figure 5.5 the power consumption of the cluster with and without the on/off policy is shown in comparison to the request rate. It can be clearly seen that shutting down unnecessary nodes effectively reduces power consumption. The total energy consumption for the test run was 175 Wh with scaling and 250 Wh without scaling, so the energy savings were 30%. In figure 5.6 the number of active servers is compared to the workload. Due to the fast replay speed of the workload trace, the booting time of offline nodes resulted in the visible lag in scaling. Because of this lag the system had to boot nodes earlier than necessary under real world conditions. It also has to be pointed out that the fast replay has a negative effect on the energy efficiency, since booting time and booting power consumption had a much larger effect. In relation to the simulation speed a worker

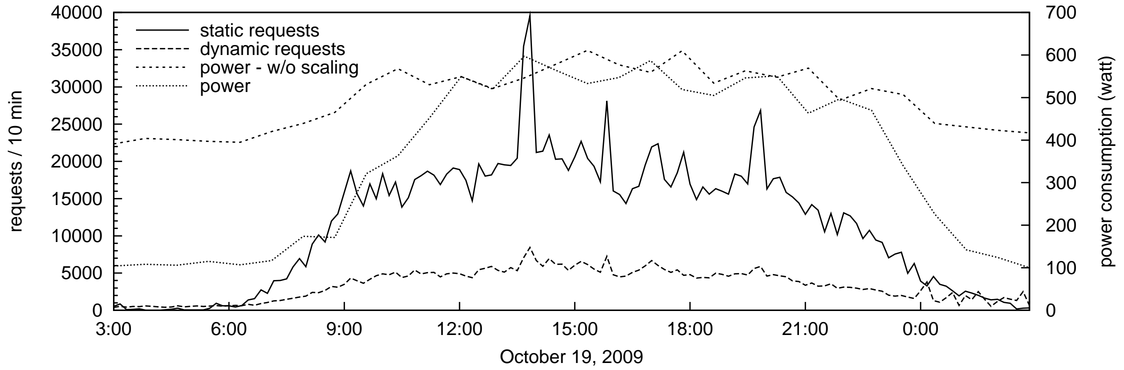



Figure 5.5.: Energy Consumption Compared to Workload

machine needed 50 minutes for booting. As mentioned before, the first day of the lecture period has the highest workload in the year. So even if the workload were equally high every day, the savings per year were 1300 kWh.

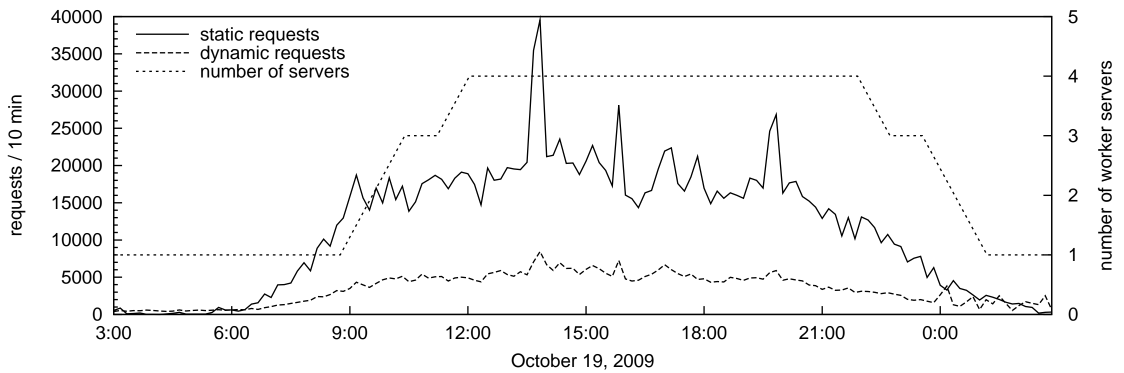



Figure 5.6.: Number of Active Servers Compared to Workload

This simple test was carried out to check the implementation costs of the Scalileo integration and to get results on the efficiency increase using autonomic scaling. It was shown that the integration of the Scalileo framework was comparatively easy and could be done within a single day. The energy efficiency results are considerable, even though the tuning was rough.

It has to be noted that the reduction of energy usage may come at the cost of increased wear of the hardware. To the best of our knowledge there is no publicly available study on this topic. However, Belam et al. try to enhance the reliability of disk systems by selectively powering off disks [33]. Their work starts from the assumption that switching of a system for a certain amount of time rather increases the lifetime instead of reducing it.

After discussing related work on scaling and energy efficiency, we will outline the inte-

gration of Scalileo in the CDBS architecture.

## 5.3. Other Scaling Frameworks

There is little research on frameworks for self-scaling. In the cloud computing world some frameworks for automatic application scaling are used. An example of a commercial system is Scalr[1]. It uses the Amazon Elastic Compute Cloud (EC2)[2] [102]. Scalr uses different Amazon Machine Images (AMI), an Amazon proprietary virtual machine, to scale and replicate applications. These AMIs are configured to run certain applications such as web servers, load balancers or database servers. Additionally the AMIs have a monitoring suite, which is comparable to the benchmarking system of Scalileo. Unlike Scalileo, Scalr is limited to the EC2 environment and cannot be run on arbitrary clusters.

Although virtualization is a viable technique for energy efficiency – an example is the Virtual Home Environment project [123] – Scalr does not aim for energy efficiency. Scalileo can also be used with virtualization techniques. The virtualization layer can be made self-scaling, similar to Scalr, but independent of the base system. The distributed application is then started on a fixed number of virtual machines (VMs) and Scalileo allocates the VMs on real systems according to the current load.

A similar service to Scalr is provided by RightScale[3]. RightScale offers a cloud management platform, that allows automatic adaption of cloud based deployments [144]. Unlike Scalr, RightScale features a multi-cloud engine that enables it to access various cloud serving systems. However, the system is also based on the scaling of instances of virtual machines and can therefore not be integrated into an application.

## 5.4. Frameworks for Energy Efficiency

Several frameworks for energy efficiency have been proposed. Petrucci et al. have presented a dynamic framework for power aware server clusters [176]. Besides an on/off-policy, they also support dynamic voltage scaling to reduce power consumption. Specialized hardware and software to measure performance and power consumption of individual machines are used. Based on the results a theoretically optimal cluster configuration is calculated using mixed integer linear programming. This involved approach makes it impossible to use self-scaling techniques as in Scalileo. Since management efficiency is an important factor besides energy efficiency [25], Scalileo has built-in performance benchmarks and makes simplified assumptions for energy consumption to reduce the setup complexity and management overhead.

A similar framework was presented by Rusu et al. [192]. It also uses dynamic voltage scaling and on/off policies, but relies – like Scalileo – on more simplified optimization schemes. However, it also requires extensive power consumption measurements and does

[1] Scalr - `http://www.scalr.net` (last visited 2011-04-15)
[2] Amazon Elastic Compute Cloud - `http://aws.amazon.com/ec2/` (last visited 2011-04-15)
[3] RightScale - `http://www.rightscale.com/` (last visited 2011-04-15)

not use self-scaling. With the use of these measurements the framework is able to improve energy efficiency in heterogeneous clusters. The benefit of considering heterogeneity was also demonstrated in [119]. In Scalileo heterogeneity is currently only considered on a performance level. By using adapted benchmarks and conditions, previous hardware efficiency measurements could be utilized as well in the choice of new nodes or nodes that have to be removed or replaced. The according benchmarks would simply return the result of the previous measurements.

On the application level additional energy savings are possible. For example, Horovath et al. have shown that using prioritization for request queuing in web servers additional to dynamic voltage scaling can lead to substantial energy savings [128]. However, these kind of optimizations are outside the scope of a scaling framework and have to be implemented in the scalable application.

On a larger scale the GREEN-NET framework [79] and Muse architecture [63] show how energy efficiency of computing grids and hosting centers can be improved. This is done using on/off policies as well as energy aware service level agreements. At the scale of data centers, energy consumption of network devices such as switches can also be considered. These devices usually do not offer any low power states and constantly communicate even if systems are idle. Therefore, switching them off further reduces energy consumption [19]. These techniques are currently beyond the scope of the Scalileo framework, but will be considered for future work.

Similar approaches for energy efficiency have been presented in specialized applications. For example, Chen et al. present a energy aware cluster of connection servers for Internet services [64]. It uses an on/off policy to adapt the number of connection servers to the number of TCP connections. Systems like this can also be implemented using the Scalileo framework.

## 5.5. Conclusion

This chapter described the architecture and design of the scaling framework Scalileo. The framework is highly configurable and allows dynamic scaling of Java applications. It is easy extensible and needs little to no changing of the target application. For evaluation purposes, we have implemented a simple web server cluster. By using an on/off policy, the system was able to reduce power consumption by 30% for a real world web server workload. Since we used an annual peak workload and sped up simulation time, it is clear that much higher percentage of savings are possible. To the best of our knowledge, the Scalileo framework is the only generic scaling framework that features a bootstrapping procedure.

In the next chapter we will show how this approach can be used to implement a self-scaling CDBS.

# 6. Autonomic Scaling for CDBSs

Using the Scalileo framework an autonomic CDBS can be realized. In this chapter we will outline the implementation of such a system. We will concentrate on the aspects of CDBSs that enable it to increase the efficiency and again the energy efficiency in particular. Furthermore, we will show how to manage the scaling of a CDBS autonomically. Following the MAPE terminology, we will give various symptoms that can be observed, their root causes and how to adapt the system accordingly. Although it is possible to automate various tasks in the CDBS configuration, we will limit the discussion to the scaling and node migration of CDBSs, which were also implemented in a prototype. These cover the aspects of self-configuration, self-optimization, and self-healing that are part of self-management (see section 4.4). The abstract symptoms that the system can exhibit are low efficiency, low performance and some kind of fault. To diagnose a certain symptom, the autonomic manager needs sensors to observe the symptom and knowledge to identify it.

## 6.1. Sensors

There are several data sources in CDBSs that can act as sensors; some are rather static; some are dynamic. These can be divided into sensors on the master, sensors on the nodes and sensors on the backend DBMSs. On the master there are the following sensors:

**Workload** All requests are initially sent to the master. It has therefore an overview of the current traffic on the system. The average number of queries per time unit is interesting as well as the composition of the workload, i.e. the read to write ratio.

**Query queue** The master schedules the queries on the backend DBMSs. For this the master has a query queue for every node. The length of the queue is an indicator of the load of a backend. If the database is not fully replicated, the ratio of the average queue sizes is an indicator for the quality of the load balancing.

**Answer time** Based on the average answer time of a backend the master can determine its performance. Obviously, the answer time is highly dependent on the queries. If the workload is very diverse the average answer time alone can be misleading and must be correlated with a workload analysis. For database systems this can be a query classification as presented in section 10.

**Network performance** Based on the delay and bandwidth, the master can evaluate the network performance.

**Node availability** The master has regular contact with the nodes; it is therefore also aware of the availability of the nodes.

On each of the backend nodes, a worker process observes the system state and the following measurements are carried out:

**System performance** If the cluster is heterogeneous, the worker can perform benchmarks to determine the performance of its host. Apart from direct benchmarks also hardware specifications such as CPU clock speed and available RAM can be analyzed. However, the performance of a DBMS is hard to predict based solely on the hardware characteristics [15]. Therefore, actual benchmarks results are preferable to the specifications, an example of such a benchmark is SysBench[1].

**System utilization** The worker can monitor the current system utilization. This includes the CPU load, the memory utilization and the I/O workload. These can be measured with OS programs like top[2] or Iotop[3].

**Disk space** An important factor for the CDBS is the available disk space on each node. This includes the available free space, as well as the space that is used by the database.

Obviously, these parameters can also be observed on the master. The DBMS itself has several interfaces that enable the user to get information about the query processing:

**Optimizer** The optimizer of the DBMS estimates the processing cost of a query. Usually, this can also be logged by the system. Papadomanolakis et al. have shown that the optimizer can be utilized to optimize the physical layout of a system [172].

**Logs** Most DBMSs feature several logs that store the queries, status information, warnings and errors. MySQL, for example, has a so-called *slow query log* that enables particularly long running queries to be tracked for special optimization.

Besides these sensors many others are possible. As mentioned above, there are special monitoring tools like Ganglia that can be used to get a more detailed view of the system. Based on the data of these sensors and its knowledge base, the manager will observe the symptoms and diagnose the problems in the system.

## 6.2. Knowledge

The mater's knowledge can come from external sources and it can be created by the master itself. Obviously, the various sensor outputs are a knowledge base that enable it

[1]SysBench: a system performance benchmark - `http://sysbench.sourceforge.net` (last visited 2011-04-15)
[2]Procps package - `http://procps.sourceforge.net/` (last visited 2011-04-15)
[3]Iotop - `http://guichaz.free.fr/iotop/` (last visited 2011-04-15)

to retrieve further information about the system behavior. Typically, the addresses of the nodes are external knowledge, even though they could be automatically retrieved using Universal Plug and Play[4] (UPnP). The symptom definitions are also external knowledge that is passed to the master. We identified the abstract symptoms low performance, low efficiency and fault earlier. These high level symptoms will not be directly observed. A concrete symptom is the overload of a backend or the complete CDBS. This can be identified from the sensor data. Based on its knowledge, the master knows how to diagnose the symptoms from the sensor data and which task must be performed to adapt the system. This can be shown in a decision tree.

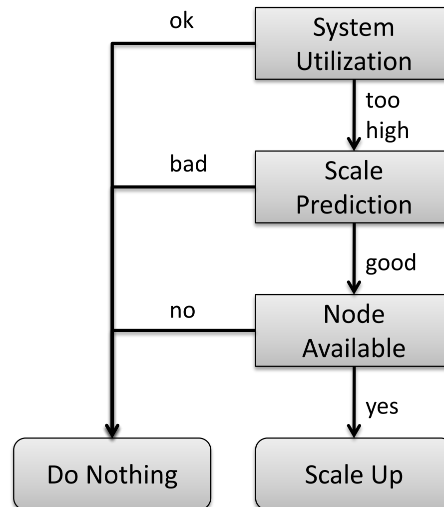



Figure 6.1.: Decision Tree for Scaling Up on High System Utilization

Figure 6.1 shows a part of a decision for the symptom *high system utilization*. In a fully replicated system, this can be identified by the average CPU and / or hard disk utilization of the backends. The actual thresholds for the utilization can be external knowledge or they can be retrieved from the sensor data. The scale prediction computes the theoretical speedup that results from the scaling. If the workload is upload heavy, a scaling might reduce the throughput. If this is the case the system will not be scaled. Finally, there is a test to establish if a node is available for scaling. Other symptoms are handled in a similar way. It is also possible to use proactive techniques this way. For example, trends in the workload can be used to predict future development. The result of the prediction can also be treated as a symptom.

Similar to *high system utilization* is *low system utilization*. If the system is frequently underloaded, the number of active nodes will be reduced. Furthermore, we treat the symptoms *lack of disk space* and *node failure*. In a fully replicated system, the lack of disk space can only be treated by migrating the database to a node with more disk space.

[4]The UPnP Forum - `http://www.upnp.org/` (last visited 2011-04-15)

If a node fails, i.e. it is not responding or only producing errors, it has to be replaced. This is the same as a migration. Yet, the database has to be transfered from another node. The high level procedures - scaling and migration - are composed of small tasks that can be executed by the effectors.

The decision trees can be built manually or using machine learning. A heuristic algorithm for generating decision trees is the C4.5 algorithm [185]. An advantage of decision trees over other models is that they are easy to understand and interpret. Therefore, new routines can be modeled easily. However, for non discrete data other classification strategies such as neuronal networks and support vector machines perform better [141]. If the benchmark results are vector-valued or real-valued a combination of decision trees and neuronal networks can be used, in which inefficient branches of the decision tree are replaced by neuronal networks [200].

## 6.3. Effectors

The manager has several ways of influencing the system. The low level effectors all have a single task, which in combination enable them to execute a high level procedure such as scaling or migration. In our prototype we use the following effectors:

**Issue command** To start the backend DBMS or issue a bulk loading operation, an effector for command execution is used.

**Transmit data** The master can send data and retrieve data from nodes. Furthermore, it can issue the transmission of data between nodes.

**Server off** A special command is used to shut a server down. Usually, this can only be issued by the super user. Therefore, we use a special configuration that allows a regular user to shut the system down.

**Server on** To wake a server up, we use the Wake-on-LAN standard [2]. The master sends a so-called magic packet to a sleeping node's network card, which then starts the server.

**Synchronize database** The synchronization of the database can be done as described in section 4.1.

Using these effectors scaling and migration can be implemented. The actual procedure depends on the state of the system at any given point. In figure 6.2 a schematic of the scaling up procedure can be seen.

When the decision to scale the system up has been made, it has to be planned and executed. The planning phase chooses which tasks have to be carried out. In the example in figure 6.2, the node for scaling has to be chosen first. This can be an arbitrarily available node; if the system is heterogeneous, it is also possible to choose a node that best suits the system. The choice can be based on the performance of the available nodes, or if a

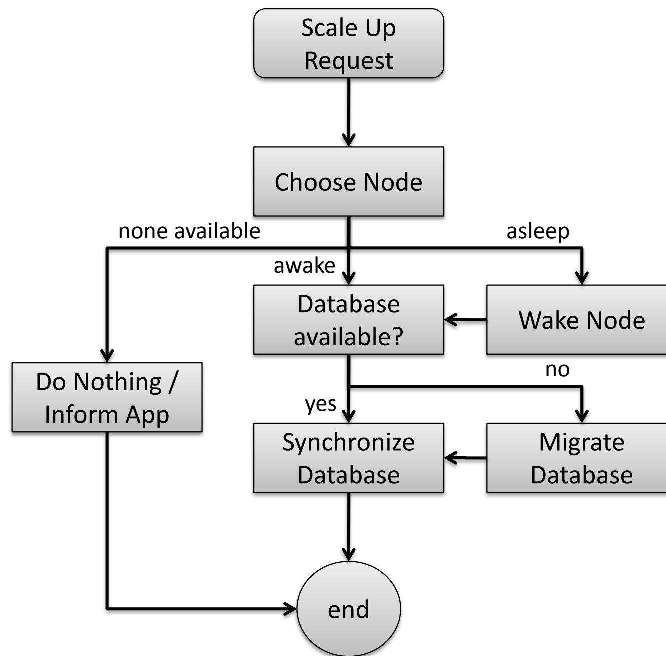



Figure 6.2.: Scaling Procedure

system already contains a DBMS and the database. The node can be either running or switched off. If it is off, it has to be waken up first; this also includes the start of the system. If no node is available, the system cannot be scaled and therefore nothing is done. If the node is part of the system for the first time, it has no instance of the database. In this case, a current snapshot will be copied onto the system. If the node already has a copy of the database, it will probably be outdated. Either way, the database has to be synchronized. If the copy is very old, it might be more efficient to replace the database with a more current snapshot. Then the missed updates have to be processed until the database is up to date. In general, the process is an optimization problem. To reduce the complexity of the choice of nodes, we use a point system as presented in section 5.1.7. Similarly, scaling down and node migration are implemented.

## 6.4. Evaluation

We implemented a prototype to give a proof of concept for autonomic scaling. As for the web server example in section 5.2, we measured the energy efficiency improvements that can be reached as a showcase for efficiency. The same hardware cluster as described in section 5.2 was used. The system architecture can be seen in figure 6.3. The system consisted of 6 nodes. One of the nodes served as controller node, which hosted the Scalileo master and the query scheduler. The query scheduler wrote an update log, in order to

allow an asynchronous synchronization of offline nodes. The backends hosted a MySQL DBMS in version 5.1.49. Again, we used the PCE-PA 6000 power analyzer to measure the energy consumption.

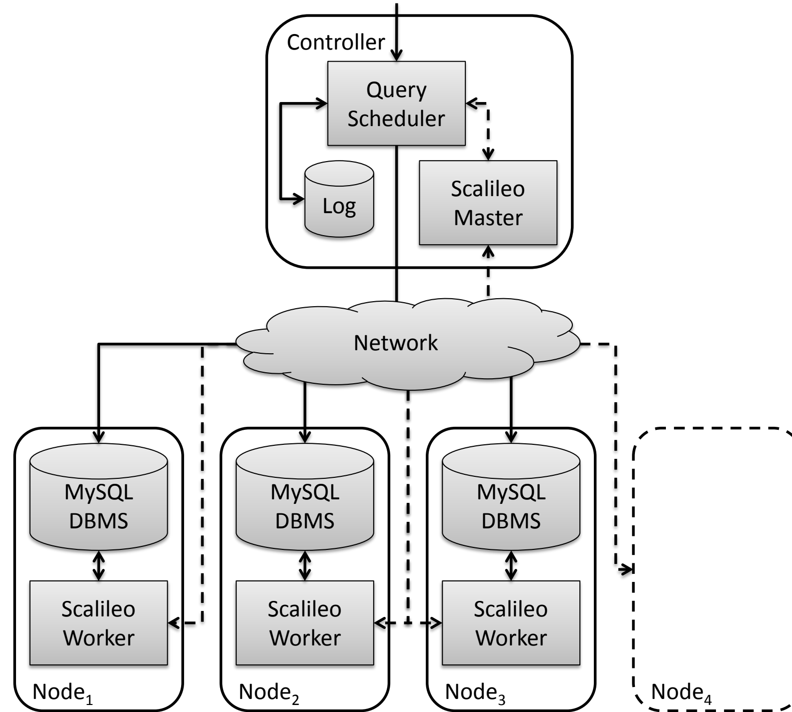



Figure 6.3.: Architecture of the Autonomic Scaling CDBS

To test a realistic database application, the TPC-App benchmark was used, which is presented in detail in section 17. The test database was generated with common scaling factor $EB = 300$, which resulted in a raw database size of 280MB. Since TPC benchmarks do not define variations in the amount of workload, we used the Stud.IP traces to generate realistic workloads. The scaling factor of TPC-App was chosen to be similar to the real Stud.IP database. For this test, we used only the dynamic HTTP accesses of the workload trace since they also create database accesses in the real system. The trace can be seen in figure 6.4.

To generate database accesses from the log, we aggregated the HTTP accesses in 2.5 minute steps. The resulting sum was scaled down by a factor of 0.01. For each request 30 transactions where generated according to the TPC-App specification. To reduce the test time, we sped up the trace replay by a factor of 30 which reduced the test time to 48 minutes. The requests were generated in 5 second steps. In the end, the average workload was 70 requests per second and the peak load was 250 requests per second. The ratio between read and update requests was 1 to 7. For each select statement 7 insert or update statements were processed. However, the aggregated workload of the select

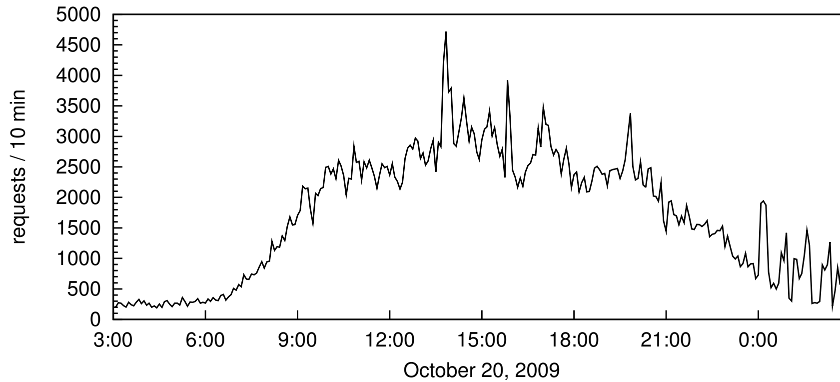



Figure 6.4.: Trace of Dynamic HTTP Accesses of Stud.IP at University of Passau for the Second Day of the Winter Term 2009

statements was 7 times higher than the workload of all updates; the ratio between the read and update workload was 0.88 : 0.12. According to the speedup formula in equation 3.14 this allows the following speedups:

| #Nodes | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Speedup | 1 | 1.8 | 2.4 | 2.9 | 3.4 |

This shows that replicating the database on five backends results in considerable performance improvements and therefore makes sense. For higher update loads this might not be the case as we explained earlier in section 3.4; this will also be shown in the evaluation of the allocation strategies in section 15. We used a single day of a periodic workload and, therefore, we initially installed all systems and loaded the database. This is consistent to a real setup, since the installation and initial data load will usually not be part of the daily routine. We ran two test series: one without scaling, using all 5 nodes; and one with scaling, starting from a single node. Because of the faster replay, booting and resynchronization have an over-proportional effect. Booting a system took 85 seconds on average, starting the DBMS 23 seconds; in real-time this would mean over 50 minutes of initialization. Additionally, after scaling the system up the new backends had to be resynchronized; at the first scaling, this took on average 5 seconds and for the 5th backend on average 55 seconds. This corresponds to up to 30 minutes of resynchronization in real time. We used reactive scaling constraints. Scaling was done based on the CPU utilization using a maximum reduction. If one backend system had over 85% load over 20 seconds a scale up request was sent. Scaling down was done if CPU utilization on one backend was over 20 seconds below 30%. Each test series was repeated 5 times. An additional constraint for inadequate answer times for queries was used, however, the CPU utilization constraint was always more precise and anticipated the result. Since the database was relatively small constraints using the I/O performance where not employed.

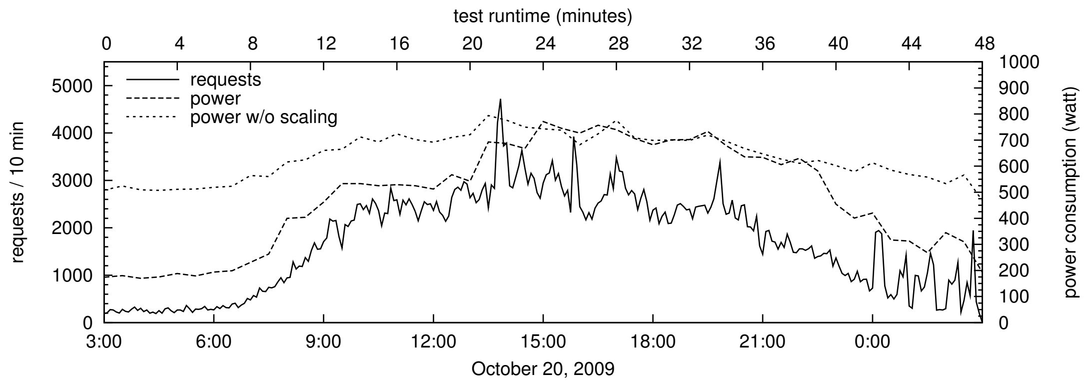



Figure 6.5.: Energy Consumption Compared to Workload

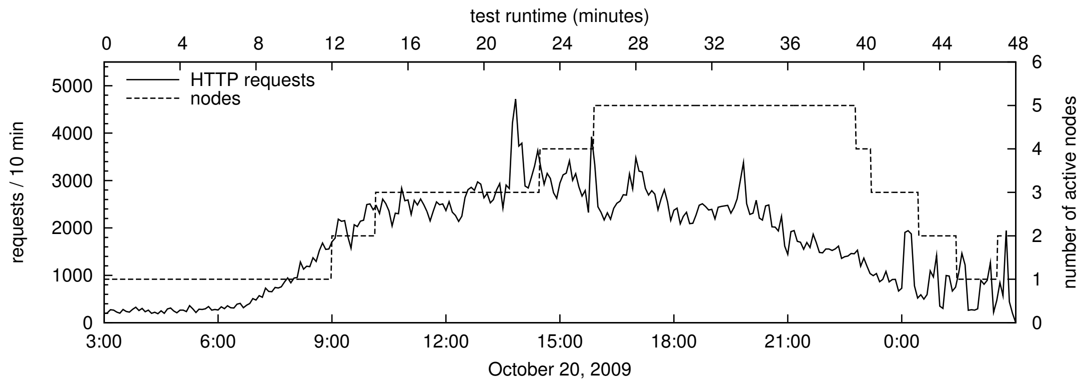



Figure 6.6.: Number of Active Servers Compared to Workload

In figure 6.5 the energy consumption can be seen. Both energy consumption without and with scaling are shown. The measurement included all backends as well as the master node. The total energy consumption was 508.3 Wh with scaling and 667.7 Wh without scaling. This results in energy savings of 160 Wh or 24%. It can be seen that for both the static configuration and the dynamic configuration the energy consumption is related to the workload. However, because of low efficiency of the systems for low utilization the dynamic configuration fits the workload curve much better. As our previous measurements have shown (cf. section 5.2) the workstations require a minimum of 91 Watts when they are idle and a maximum of 200 Watts while booting. Hence, resulting in a base consumption of at least 50%. This is the main source for improvements using the scaling.

In figure 6.6 the number of active nodes for the dynamic configuration can be seen. A node is identified as active if it is able to handle SQL requests and booting and initialization are already finished. Due to the conservative constraints, the system is further scaled when the peak load is already reached. It has to be noted, however, that the according scaling requests have been sent up to 2.5 minutes earlier (75 minutes in real time). This

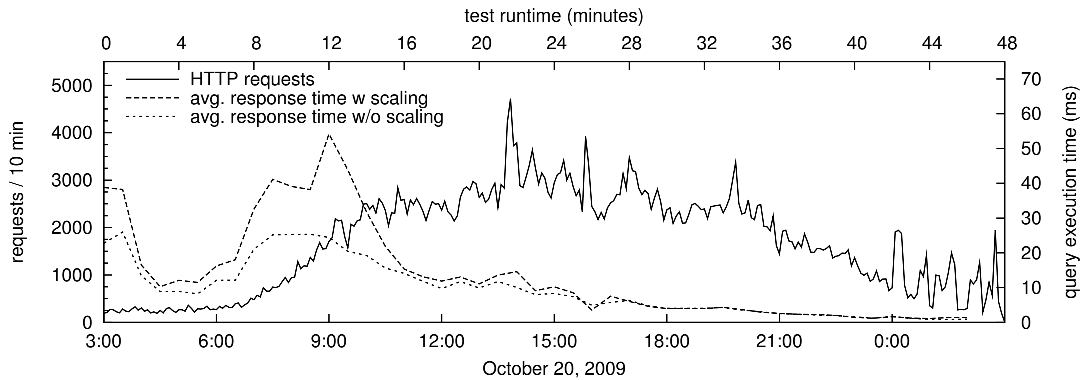



Figure 6.7.: Average Response Time Compared to Workload

results in a considerable lag in the scaling procedure. Furthermore, it can be seen that at the end of the trace the system was scaled up again due to stronger variations in the workload. An improvement would be the possibility to stop a scaling process if it is not needed any more.

In figure 6.7 the average query response times can be seen for a cold start of the cluster. It can be seen that both the static and the dynamic configuration have a noticeable initialization overhead, which has more effect for the single system than the replicated system. This overhead stems from increased query answer times due to cache misses while the update times are relatively constant. The answer time of both configurations have a similar evolution until the load becomes too high for the single system. When the second and third nodes are active in the dynamic configuration the response time starts converging the response time of the static configuration. Although the dynamic configuration has an increased response time the average response time never exceeds 50 milliseconds and is on average 10 milliseconds. A very important detail that can be seen in the figure is that the scaling procedure – as implemented in our prototype – does not decrease the throughput.

The tests show that a scale-out approach can be used in the context of cluster database systems. Furthermore, it can be seen that considerable energy savings can be achieved using a simple on/off policy. The use of the Scalileo framework reduced the implementation overhead to a minimum. Apart from the energy savings, the system is very cost effective since it is completely built of off-the-shelf hardware. Although we used a relatively small database in the test, the fast replay and the resulting additional costs for booting, initialization, and synchronization indicate that for real systems much larger databases can be used with similar results.

## 6.5. Research Projects in Autonomic Scaling

Although there is a large body of research on autonomic tuning of database systems [195, 198, 50], there is little work on autonomic scaling of clustered database systems. Many systems do not allow a new node to be added at runtime [58]. We are aware of only two other systems that introduce autonomic scaling of distributed DBMSs. Some work has been done on estimating the right scale of a distributed database [98, 111], which can help to make the scaling proactive.

### 6.5.1. Ganymed

As mentioned in section 3.5, the Ganymed project allows dynamic creation of satellite databases. This is, however, not an autonomic procedure, but on very update heavy loads, the system automatically delays the propagation of updates and basically reduces itself to a single node system. This is actually not real scaling, even if it is a similar procedure.

### 6.5.2. KNN Prediction

Chen et al. propose a proactive scaling approach that is based on the K-nearest-neighbors (KNN) algorithm [65]. The system allows autonomic adding and removal of new DBMS nodes. The removal is done reactively if the system is underloaded. The authors pay special attention to oscillation effects, such as the constant adding and removal of nodes. Therefore, the removal is two phased, first the system is temporarily removed, but still updated and if the performance of the remaining nodes is still adequate, the node is completely removed. The adding of the nodes is done proactively, in contrast to their earlier work [205]. Based on the system metrics, as defined above in section 6.1, KNN classifiers are trained. The authors used a cluster with 8 nodes and tested it with the TPC-W benchmark (a predecessor of TPC-App). Because of the lack of variation in the amount of workload the authors used a sine function to simulate variations. With this relatively simple setting, the authors achieve good efficiency while maintaining a predefined query latency.

### 6.5.3. Sprint

Sprint is a distributed in-memory DBMS [54]. Although it does not directly support scaling, it features automatic recovery from failures. This is similar to node migration. Sprints architecture can be seen in figure 6.8. It consists of three types of logical servers. Edge servers, data servers and durability servers. These can be hosted on the same or different physical servers. Edge servers correspond to the master node in the CDBS architecture. They schedule the queries to the data servers. Data servers run in-memory database systems and execute the queries. Durability servers write logs to disk and handle the recovery. Sprint offers some form of distributed query processing. However, the queries are completely predefined and parameterized, which allows a simple table based substitution. Sprint is also capable of working with distributed data. However,

distribution is also a manual task. Despite the recovery, the system does not do an autonomic optimization of the degree of replication and the like.

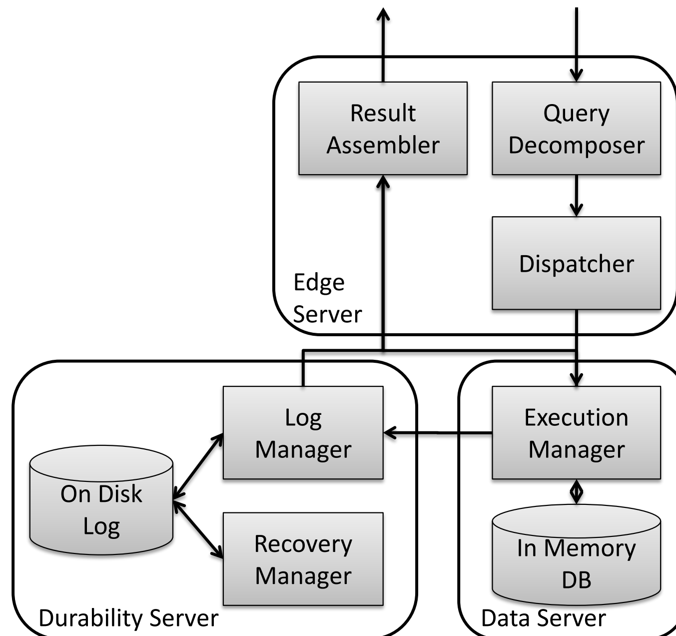



Figure 6.8.: Sprint Architecture

### 6.5.4. WattDB

WattDB is a distributed, energy proportional database system, recently presented by Härder et al. [117, 196, 131]. It has a three tier architecture consisting of data nodes, computation nodes, and a master, similar to the MySQL Cluster architecture (see section 3.5.4). The master node is the centralized coordinator for performance tuning, energy tuning and node adding and removal. Data nodes store the database files and feature only simple access routines such as selection and projection. The computation nodes perform CPU intensive operations such as joins. Energy efficiency is increased by dynamically adding and removing nodes based on the workload off the system. The system uses lightweight hardware that has reduced energy consumption. To this end, there is no information on what cost models or workload predictions are used for energy tuning. The system is implemented as a testbed for energy efficient hardware and algorithms for database systems as presented in [105, 219].

## 6.6. Discussion

There are several database system projects that incorporate autonomic scaling. In contrast to our work these all use scaling to improve one aspect of the system. Using the generic Scalileo framework, our system can be adapted to employ scaling for various optimizations. In the following table a comparison of the presented projects can be seen.

| Project | Scope | Processing model |
|---|---|---|
| Ganymed | Performance | CDBS |
| KNN Prediction | SLA | CDBS |
| Sprint | Fault tolerance | PDBS |
| WattDB | Energy efficiency | PDBS |
| CDBS | Adaptable | CDBS |

The table shows the scope of the scaling in each of the systems and the query processing model. It can be seen that various goals can be pursued with autonomic scaling. Ganymed uses scaling to increase the performance of the database system. The approach by Chen et al. focuses on service level agreements, in order to keep answer times within a certain threshold. Sprint uses a limited form of scaling for fault tolerance and WattDB increases energy efficiency by scaling. With our generic approach the system can be adapted to all of the presented goals. In contrast to our work, Sprint uses intra-query parallelism to improve performance which is traditionally a feature of parallel database systems (PDBS). WattDB has a two-staged processing model which uses distributed query processing.

In conclusion, there are some interesting approaches to refine our scaling approach using proactive techniques and more advanced analysis. However, we are not aware of any other system, that is capable of a cold start scaling of distributed DBMSs. In the next section, we will discuss advanced allocation strategies that improve disk utilization and performance in the presence of updates.

# Part III.

# Allocation

# 7. Distributed Database Layout

In chapter 3 we introduced the ANSI/SPARC reference model. The model ensures that the database can be seen as a single integrated entity. The benefit is transparent access to the database regardless of the internal representation. For integrated distributed database systems, this model can be adapted. Obviously, it is preferable, if the database can be seen as a single entity and transparently accessed. The database will therefore have an integrated schema that does not consider the distribution of data. This is illustrated in figure 7.1, for each node of the distributed system there is a local internal and conceptual schema. For the complete system a global conceptual schema is defined, as well as the external schemata. If the system is fully replicated, as we assumed before, the local and global conceptual schema are identical.

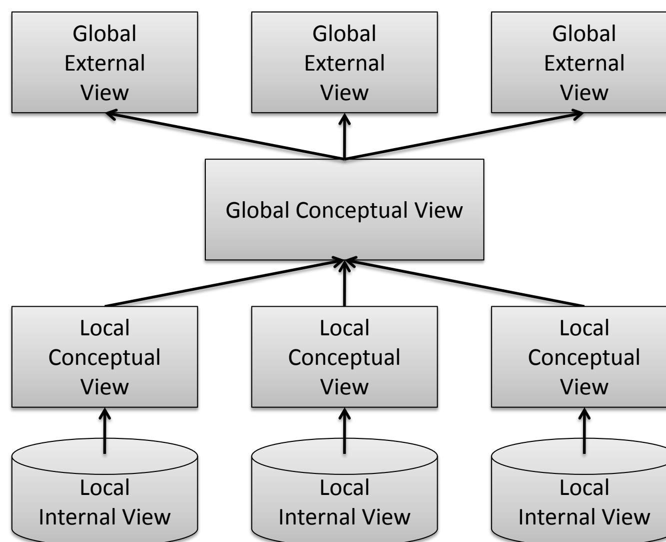



Figure 7.1.: Adaption of the ANSI/SPARC Reference Model for Distributed Systems

As we mentioned before, it is not always preferable to store the complete database on every node. The two major reasons are disk usage and update replication. Every copy of a database increases the disk usage linearly. At times of clusters with thousands of nodes, this is not feasible. As shown in section 3.4, with full replication the ratio of updates has increasing influence on the system performance in relation to the number of nodes. In order to store the data efficiently on an arbitrary number of nodes, the data must either be divided in disjoint parts or replicated or both. The data distribution must

allow efficient processing of all incoming queries. Queries usually can be processed most efficiently on a single node. Therefore, queries benefit from replication, while updates have to be processed on all nodes that contain referenced data. If the rate of updates is significant, a fully replicated system will suffer considerable performance losses. To avoid this the data can be *partitioned* such that nodes contain only parts of the database that still allow local processing of queries. Therefore, the best performance is in most cases achieved by a combination of partitioning and replication.

The global conceptual schema of a distributed database is extended by a partitioning schema and an allocation schema. The partitioning schema stores the way in which global relations are partitioned and the allocation schema stores the way in which the partitions are allocated on the nodes. In this part of the thesis, we will discuss integrated allocation strategies for the CDBS query processing model. Our goal is to develop an automatic and computationally feasible allocation strategy. The following requirements have to be met:

**Automation** The allocation has to be computed completely automatically. There should be no need for manual interference or adjustment.

**Computability** The approach has to be feasible for realistic problem sizes. In general, the allocation problem is NP-complete [193], therefore, heuristic approaches are needed.

**Efficiency** The resulting data layout should be near to the optimum based on the CDBS processing model. An optimal allocation enables an optimal throughput and minimizes disk requirements.

In this chapter we will explain the fundamentals of the distribution of relations. We will first give a short overview of different forms of partitioning and replication and after that describe the physical distribution of data fragments. The rest of part III is organized as follows. We will give an overview of related work in chapter 8. After the discussion of automatic allocation in chapter 9, we will describe the query classification for different levels of partitioning in chapter 10. We will then explain our algorithms for allocation in detail; chapter 11 explains the allocation for read-only databases; chapter 12 explains the allocation for databases with updates and chapter 13 shows extensions of the algorithms for high availability. After that, we will show how allocation can be implemented in an existing system in chapter 14. This part concludes with an evaluation of the presented methods in chapter 15.

## 7.1. Partitioning

Partitioning or fragmentation is the process of dividing a relation into several, usually disjoint *fragments*. There are two basic forms of partitioning: *vertical* and *horizontal* partitioning. Vertical partitioning divides a relation by its columns (see figure 7.2), while horizontal partitioning divides it by its rows (see figure 7.3). Either way the relation is partitioned, it must be possible to retrieve the original data from the partitioned data. This can be ensured if the following three properties are met [169]:

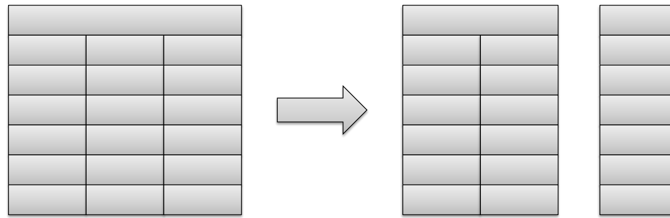


Figure 7.2.: Schematic of Vertical Partitioning

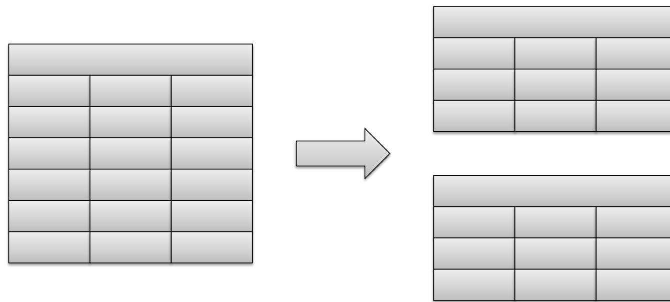


Figure 7.3.: Schematic of Horizontal Partitioning

**Completeness** All data units are assigned to a partition.

**Reconstruction** The original relation can be retrieved from the partitions.

**Disjointness** The partitions do not overlap.

Although disjointness is usually required by textbooks (e.g. [136, 169]), it is not necessary and for vertical partitioning not possible. In order to reconstruct a vertical partitioned table, every partition has to replicate a key (we will discuss exceptions to this rule below). Relations can be partitioned in three different ways, *vertically*, *horizontally* or both.

### 7.1.1. Vertical Partitioning

Vertical partitioning divides a relation into several relations with fewer attributes, but usually with an equal amount of tuples. This is depicted in figure 7.2. In order to guarantee the reconstruction of a relation, each partition requires a key of the relation. Usually, the primary key is therefore replicated in each partition. Consider the following example from section 3.1.

| ORDERS | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 1 | Screws | Mike | 11-10-2010 |
| 2 | Nails | Andy | 22-11-2010 |
| 3 | Hammer | Chris | 22-11-2010 |
| 4 | Nuts | Mike | 05-12-2010 |
| ... | ... | ... | ... |

The relation $ORDERS$ will be vertically partitioned in two parts. In order to guarantee the reconstruction of the relation, the primary key $ID$ is replicated in both partitions. Since relation names have to be unique, the new partitions names are extended with a suffix or prefix.

| $\text{ORDERS}_1$ | |
|---|---|
| ID | ITEM |
| 1 | Screws |
| 2 | Nails |
| 3 | Hammer |
| 4 | Nuts |
| ... | ... |

| $\text{ORDERS}_2$ | | |
|---|---|---|
| ID | CUSTOMER | DATE |
| 1 | Mike | 11-10-2010 |
| 2 | Andy | 22-11-2010 |
| 3 | Chris | 22-11-2010 |
| 4 | Mike | 05-12-2010 |
| ... | ... | ... |

The original relation $ORDERS$ can be reconstructed by joining the two partitions on the primary key.

$$ORDERS = ORDERS_1 \bowtie ORDERS_2 \tag{7.1}$$

In the original relational model a relation can be interpreted as a set of tuples [70]. Therefore, the DBMS does not have to guarantee the order of the tuples within a relation. However, some DBMSs keep the tuples strictly ordered by their entry. Therefore, they do not require a key on each partition, but can use the order for the join. An example is the C-Store project [211] and its commercial continuation Vertica [8].

### 7.1.2. Horizontal Partitioning

Using horizontal partitioning, a relation can be divided into several parts with equal schema. One way to do this is by applying some sort of constraint to an attribute and dividing the relation into two parts: one that satisfies the constraint and one that does not. Another way is using a distribution function that assigns tuples to a partition. Consider the following example, relation $ORDERS$ from section 3.1 will be partitioned by the constraint $DATE$ <= ’30-11-2010’.

| ORDERS$_1$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 1 | Screws | Mike | 11-10-2010 |
| 2 | Nails | Andy | 22-11-2010 |
| 3 | Hammer | Chris | 22-11-2010 |

| ORDERS$_2$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 4 | Nuts | Mike | 05-12-2010 |
| ... | ... | ... | ... |

To reconstruct the original table the partitions have to be merged:

$$ORDERS = ORDERS_1 \cup ORDERS_2 \tag{7.2}$$

Obviously, many different constraints and functions can be used. Generally, seven forms can be distinguished: round-robin partitioning, random partitioning, hash partitioning, range partitioning, list partitioning, predicate-based partitioning, and derived partitioning [86, 169, 164]. These can be divided into value-based and non-value-based partitioning. Non-value-based approaches are round-robin and random partitioning. These do not consider the values of a tuple but its position in the relation. In contrast to that the value-based partitioning approaches divide a relation according to the values of an attribute.

**Round-Robin Partitioning** Round-robin partitioning distributes the tuples evenly on the partitions. This is done using a modulo of the row number of a tuple. This partitioning is mainly used in parallel database systems. Obviously, each query has to be executed on each partition. However, the reduced size of the partitions reduces the execution time. For two partitions the round-robin partitioning of $ORDERS$ would be:

| ORDERS$_1$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 1 | Screws | Mike | 11-10-2010 |
| 3 | Hammer | Chris | 22-11-2010 |
| ... | ... | ... | ... |

| ORDERS$_2$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 2 | Nails | Andy | 22-11-2010 |
| 4 | Nuts | Mike | 05-12-2010 |
| ... | ... | ... | ... |

**Random Partitioning** Similar to round-robin partitioning, random partitioning aims to distribute tuples evenly across the partitions. Using a random function the tuples are assigned to the partitions without any correlations which might appear in a round-robin approach.

**Hash Partitioning** Hash partitioning uses some form of hash function on an attribute of the relation and partitions the relation accordingly. This is similar to the random approach. However, for exact match queries on the hashed attribute, the hash function can be used to find the matching partition. This enables to answer a query by accessing only one partition.

**Range Partitioning** As depicted in the example above, range partitioning divides a relation according to value intervals of a particular attribute. This is probably the most common used form of horizontal partitioning. This form is especially useful for queries with high locality. The partitioning constraints can be added to the query attributes and if the query results are empty, the query does not have to be executed. Consider the following example SQL query:

Listing 7.1: SQL Query Example

```
select CUSTOMER, sum(ITEM)
  from ORDERS
 where DATE >= date '2010-10-01'
   and DATE <= date '2010-10-31'
 group by CUSTOMER
```

If the query is extended with the range constraint from above, it is clear that only the first partition can contain matching tuples. Using this information the query execution can be reduced to a single partition and hence in a distributed DBS to a single node. Obviously, this is only possible if the attribute in the range constraint is also an attribute of the predicate part in the query. The quality of a range partitioning greatly depends on the choice of the partitioning intervals; for poor choices the result may be data skew and execution skew. Data skew is an uneven distribution of the data, where one partition has a much larger part of the data than another. Similarly, execution skew is an uneven distribution of workload across partitions.

**List Partitioning** Another option to partition relations according to their attribute values is list partitioning. A list of attribute values is assigned to each partition. For the relation orders this could be $L_1 = \{\text{Screws}, \text{Nuts}\}$ and $L_2 = \{\text{Hammer}, \text{Nails}\}$ for the attribute $ITEM$.

**Predicate Based Partitioning** As well as constraints that divide the relation into several intervals, the relation can be divided based on predicates. This is similar to range partitioning, except that predicates not intervals are defined. The relation is then partitioned

into tuples that satisfy the constraint and tuples that do not. An example could be the constraint:

```
ORDERS.CUSTOMER = 'Mike'
```

The resulting partitions are:

| $ORDERS_1$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 1 | Screws | Mike | 11-10-2010 |
| 4 | Nuts | Mike | 05-12-2010 |
| ... | ... | ... | ... |

| $ORDERS_2$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 2 | Nails | Andy | 22-11-2010 |
| 3 | Hammer | Chris | 22-11-2010 |
| ... | ... | ... | ... |

If the predicates are retrieved from the query workload, the locality and load balancing can be adapted automatically.

**Derived Partitioning** A special form of horizontal partitioning is derived partitioning. If a partitioned relation is frequently joined with another relation, the other relation can be partitioned according to the join predicate. Consider the relation $CUSTOMERS$ from section 3.1. The relation $ORDERS$ can be partitioned using the foreign key $CUSTOMER$ on $CUSTOMERS.NICK$. If each value of $CUSTOMER$ is in a separate partition the result is:

| $ORDERS_1$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 1 | Screws | Mike | 11-10-2010 |
| 4 | Nuts | Mike | 05-12-2010 |
| ... | ... | ... | ... |

| $ORDERS_2$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 2 | Nails | Andy | 22-11-2010 |
| ... | ... | ... | ... |

| $ORDERS_3$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 3 | Hammer | Chris | 22-11-2010 |
| ... | ... | ... | ... |

If the partitioning has to be disjoint, the referenced attribute must be either unique or all instances with the same value must be in the same partition.

### 7.1.3. Hybrid Partitioning

Obviously, multiple partitioning approaches or constraints can be used simultaneously. Relations can be partitioned horizontally and vertically, or with multiple horizontal approaches, or both. The different forms of partitioning can be applied *equally* or *hierarchically*. For equal partitioning, multiple partitioning approaches are applied to a relation and all resulting partitions iteratively. Using hierarchical partitioning, the resulting partitions can be partitioned using different approaches or constraints. Hence, the execution order influences the resulting partitions. This is shown in the following examples.

**Equal Multidimensional Horizontal Partitioning** In practice usually only predicate partitioning and range partitioning are used on multiple attributes. They are especially useful if multiple attributes are referenced with equal frequency or if a single attribute does not allow a sufficiently fine-grained partitioning. Consider the following example, in which relation orders are partitioned according to the predicates $ITEM$ = 'Screws' and $CUSTOMER$ = 'Mike'. In general the result is a partitioning into four fragments with the predicates:

- $(ITEM = \text{'Screws'}) \wedge (CUSTOMER = \text{'Mike'})$
- $(ITEM = \text{'Screws'}) \wedge \neg(CUSTOMER = \text{'Mike'})$
- $\neg(ITEM = \text{'Screws'}) \wedge (CUSTOMER = \text{'Mike'})$
- $\neg(ITEM = \text{'Screws'}) \wedge \neg(CUSTOMER = \text{'Mike'})$

There is no entry that satisfy the predicate $(ITEM = \text{'Screws'}) \wedge \neg(CUSTOMER = \text{'Mike'})$, hence the result are three partitions:

| ORDERS$_{(ITEM=\text{'Screws'})\wedge(CUSTOMER=\text{'Mike'})}$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 1 | Screws | Mike | 11-10-2010 |
| ... | ... | ... | ... |

| ORDERS$_{(ITEM=\text{'Screws'})\wedge\neg(CUSTOMER=\text{'Mike'})}$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 4 | Nuts | Mike | 05-12-2010 |
| ... | ... | ... | ... |

| ORDERS$_{\neg(ITEM=\text{'Screws'})\wedge\neg(CUSTOMER=\text{'Mike'})}$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 2 | Nails | Andy | 22-11-2010 |
| 3 | Hammer | Chris | 22-11-2010 |
| ... | ... | ... | ... |

**Hierarchical Horizontal and Vertical Partitioning** Hierarchical partitioning is interesting if different parts of a relation are accessed differently. For example, the treatment of 'Nails' and 'Hammer' may differ to that of 'Screws' and 'Nuts' in the $ORDERS$ relation, in such a way that for 'Nails' and 'Hammer' only the quantity of orders is interesting, while for 'Screws' and 'Nuts' the complete relation is accessed. Hence, the relation is list partitioned accordingly and then the Nails and Hammer partition may be further partitioned vertically into relations {[ID, ITEM]} and {[ID, CUSTOMER, DATE]}:

| ORDERS$_{ITEM=\{\text{'Screws'},\text{'Nuts'}\}}$ | | | |
|---|---|---|---|
| ID | ITEM | CUSTOMER | DATE |
| 1 | Screws | Mike | 11-10-2010 |
| 4 | Nuts | Mike | 05-12-2010 |
| ... | ... | ... | ... |

| ORDERS$_{ITEM=\{\text{'Nails'},\text{'Hammer'}\}1}$ | |
|---|---|
| ID | ITEM |
| 2 | Nails |
| 3 | Hammer |
| ... | ... |

| ORDERS$_{ITEM=\{\text{'Nails'},\text{'Hammer'}\}2}$ | | |
|---|---|---|
| ID | CUSTOMER | DATE |
| 2 | Andy | 22-11-2010 |
| 3 | Chris | 22-11-2010 |
| ... | ... | ... |

There are many different methods and algorithms to compute a partitioning of a relation. We will give an overview in chapter 8.

## 7.2. Replication

Although partitioning can speed up the query processing of a single node system [104], it is frequently used in distributed systems to improve disk utilization and performance. Until now, we have only considered the full replication of the database. However, the data can also be fully distributed or partially replicated. The full replication and the full distribution of a database are only applied in special cases [169]. Full replication can only be used if the database is small and the number of nodes is limited. The full distribution can be found in parallel databases (e.g. round-robin or random partitioning). Larger distributed systems will therefore usually use a partially replicated database layout. The allocation decides where the copies of a partition are placed.

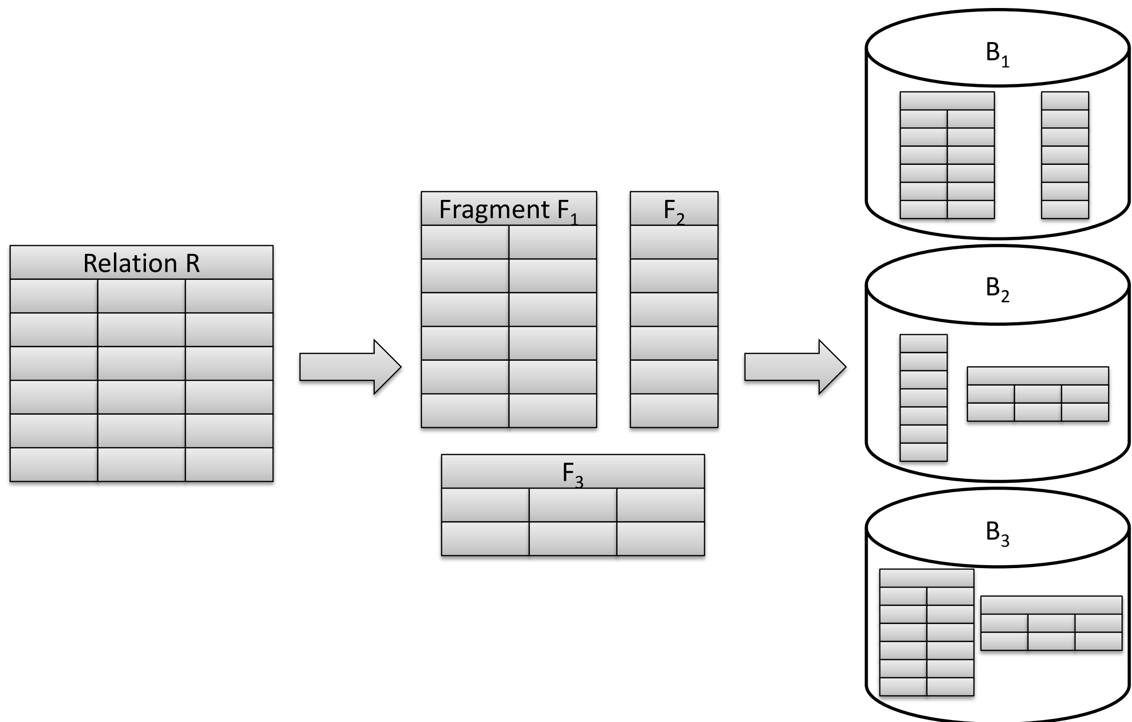



Figure 7.4.: Partitioning and Allocation

## 7.3. Allocation

Allocation is the process of assigning and placing data fragments $F_1, \ldots, F_m$ on the nodes or backends $B_1, \ldots, B_n$ of the distributed system. This is shown in figure 7.4. An optimal allocation is dependent on the data access, the available disk space, the disk cost, the maximum disk space, the transmission cost, etc. The general problem has been extensively studied and originates from the file allocation problem in computer networks [68]. This and many variations have been proven to be NP-hard [193]. For cluster database systems, the problem can be reduced, since queries are processed locally and hence transmission costs are negligible. An optimal allocation is therefore a minimal allocation that allows local processing and achieves maximum throughput. Obviously, further constraints have to be considered. We will distinguish two instances of the allocation problem: the read-only case, in which the workload contains no updates, and the update-considering case. Often, the allocation is a manual task and considered as black art [169, 223]. However, in large scale systems manual approaches are not feasible, therefore, we will present an automatic approach in chapter 9. Before that, we will give an overview of related work on partitioning and allocation.

# 8. Related Work

There is a plethora of work on data placement and allocation in database systems. Although data placement is a mature field of research and has been studied for over 40 years, the ever-growing number of publications indicates that there is a constant need for improvements. As mentioned above, the general problem is based on the file allocation problem in distributed computers (FAP), which was described in 1969 by Chu [68]. FAP and probably all of its descendants are at least NP-complete [101]. As we will show in section 11.3, this is also true of the allocation problems described in this thesis. In this chapter, we will classify different approaches to the allocation problem and discuss selected representatives of these approaches. Due to our processing model we will concentrate on shared nothing systems; however, we are aware that there are many valid approaches for other systems such as shared disk or single node.

In general, two categories of allocation approaches can be found, static and dynamic approaches. While static approaches try to find a good initial allocation, dynamic approaches try to optimize a given allocation by reorganizing it during execution. This classification is not distinct, since many approaches have procedures for both. The allocation process has multiple steps. Obviously, each of the of these steps can be implemented differently. Hence, a further classification can be made based on the steps of an allocation described by Daudpota [83]:

- Collect existing global relations.
- Analyze Frequently Asked Queries.
- Set data allocation objectives.
- Transform global relations into fragment relations.
- Allocate fragment relations to sites.

We will exclude the collection of the global relations, since this is usually no problem and does not change the overall procedure. A similar classification was presented by Zhou and Williams [231]. Although they used the allocation objectives as general classification, they remarked that algorithms differ in their form of declustering, data placement and reorganization. Based on these two categorizations, we will analyze related approaches according to the following characteristics:

**Analyzed Workload** Generally, the allocation is performed on the basis of workload analysis. Two basic variations are common: the analysis of transaction and the analysis

of single queries. Static approaches in the design phase of a database often require estimations of the future workload characteristics. However, the form of the collection of common queries usually is not an integral part of the algorithm.

**Data Allocation Objectives** The most common objective in data allocation is performance optimization. This can be found in two different forms: throughput optimization and response time optimization. However, there are multiple other optimization goals, such as storage balancing, network traffic minimization, etc.

**Partitioning** There are many possible ways to partition data. In distributed systems one goal of the partitioning is to generate multiple fragments of the data with high locality of the accesses.

**Data Allocation** The allocation itself is in general computed heuristically. However, there are also optimal approaches. The algorithms use very different models to generate an allocation. We will report on the relevant allocation approach and if it interacts, how it does with the other steps in the process.

**Reorganization** Since workloads in distributed systems are likely to change, most integrated allocation strategies include a reorganization strategy that allows the allocation to be adapted.

There are several approaches that only solve single aspects of the complete approach. For partitioning in particular there are many different algorithms, some of which are intended as a part of an allocation, while others try to improve the data layout of single node systems. We will give an overview of these algorithms, since they often serve as the basis for integrated approaches. In the following we will first give an overview of selected partitioning algorithms; after that we will discuss allocation-only algorithms briefly. The main part of this chapter will then discuss selected integrated approaches to the allocation problem.

## 8.1. Partitioning

Most partitioning algorithms concentrate on a single type of partitioning. Different algorithms can then be combined to hybrid partitioning algorithms. Horizontal partitioning algorithms are usually based on the evaluation of the predicates of frequently asked queries. An exception are round-robin, hash and random partitioning for parallel systems [86, 36, 189, 207, 93, 103]. These use full or partial declustering of relations in order to increase the intra-query parallelism [153].

General formulations of predicate based approaches can be found in many textbooks such as [169, 136, 81, 61]. Therefore we will only give a short summary. In general the predicate based approach can be divided into three steps:

- Find predicates

- Build combinations
- Recombine fragments / reduce predicates

In the first step simple predicates of all or frequently asked queries are gathered, where simple predicates are triples consisting of an attribute of a relation, a relational operator and constant which should lie in the domain of the attribute. An example is:

```
ORDERS.CUSTOMER = 'Mike'
```

Each of these simple predicates can be inverted and combined into minterm-predicates. If $P = \{p_1, \ldots, p_n\}$ is the set of all predicates, and $p^-$ is the inversion of predicate $p$. The set of all possible minterm-predicates is $M = \{m|m = \bigwedge_{i=1}^{n} p_i^{\pm}, p \in P\}$ [81]. $M$ is a complete, non-overlapping horizontal partitioning. However, the cardinality of $M$ is $2^n$. Since many simple predicates can usually be found, the set of minterm-predicates has to be reduced. When and in which way the reduction is done allows a given algorithm to be classified. A direct implementation of the general formulation above was presented by Ceri et al. [62]. Iteratively simple predicates are added to the partitioning. The resulting minterm-predicates are reduced, by first sorting out contradictory minterm-predicates and minterm-predicates that are irrelevant. An example of a contradictory minterm-predicate is the following:

```
ORDERS.CUSTOMER = 'Mike' AND ORDERS.CUSTOMER = 'Andy'
```

A minterm-predicate with a irrelevant part is:

```
ORDERS.CUSTOMER = 'Mike' AND NOT ORDERS.CUSTOMER = 'Andy'
```

The relevance of each simple predicate is tested. This is the case if the resulting partitioning results in a significant improvement. This can be analyzed using the access frequency access($m$) of a minterm-predicate and the size of the resulting fragment size($f$). The access frequency is the number of requests containing the relevant predicates. A predicate $p$ is relevant for a minterm-predicate if the resulting partitions have different access rates. This can be formulated as:

$$\frac{\text{access}(m \wedge p)}{\text{size}(f_p)} \neq \frac{\text{access}(m \wedge p^-)}{\text{size}(f_{p^-})} \tag{8.1}$$

Obviously, this approach still leads to a huge set of minterm-predicates and partitions. Therefore, other approaches do not initially collect all simple predicates, but only the most common.

Another approach to the combination of predicates is the use of graphs. Navathe et al. represent the affinity of predicates, i.e. the frequency in which they are used within a single request or transaction by an edge with according weight between the predicates [163]. Curino et al. use tuples as nodes and then connect nodes which are used together in the same request or transactions [80]. Another approach based on hypergraphs was presented by Koyutürk and Aykanat [142]. In all three cases a partitioning can be found using a graph partitioning algorithm.

For vertical partitioning two basic approaches exist: splitting and grouping [169]. Splitting algorithms start with a complete relation and then iteratively partition it, while grouping algorithms start with a complete decomposition of a relation and iteratively join the fragments. An example of a splitting algorithm was presented by Navathe et al. [162], which is an extension to the method presented by Hoffer and Severance [126]. The algorithm uses the affinity of attributes and then clusters them in order to find a good partitioning. A grouping algorithm for single node systems was presented by Hammer and Niamir [115]. This approach was later extended by Sacca and Wiederhold for distributed architectures [193].

The algorithms presented all use access frequencies as the basis of the partitioning. However, it has been shown that this is inappropriate since it neglects the actual cost of a query [170, 172]. Several approaches extend the algorithms above with the usage of cost models and cost analysis [78, 171]. Overviews of further vertical partitioning algorithms can be found in [160, 233].

As we will show in section 10, our approach to partitioning is not directly comparable to the algorithms presented above since it is integrated into the allocation. However, the predicate-based classification is similar to the predicate-based partitioning. In contrast to the approaches presented, we use a cost function instead of the access frequency. Our attribute based classification leads to a vertical partitioning that is comparable to a grouping approach.

## 8.2. Allocation

As mentioned above, the allocation algorithm can have multiple goals, such as performance optimization, reliability, etc. Özsu and Valduriez present a general problem definition which respects various factors such as network characteristics and properties of the host systems [169]. The resulting optimization problem is NP-complete and therefore not feasible for realistic problem sizes. There have been many modifications of the model, for example, for sites in communication networks [26] and for special purposes such as heterogeneous storage [55, 40] and energy-efficiency [139]. To compute an allocation for realistic problem sizes heuristics are used. These either exploit the relation to the bin packing problem and apply first-fit and best-fit strategies [52, 193], or use metaheuristics such as simulated annealing [24] or evolutionary strategies [14].

Our allocation approach is also presented as an optimization problem in section 11.2. In contrast to other allocation strategies (e.g. [169, 26]), we do not consider network traffic, since we aim for local execution and consider cluster hardware configurations rather then wide area networks. Because of the NP-completeness, we use a best-fit strategy to find a initial solution and a evolutionary algorithm to improve the initial solution. However, the independent calculation of partitioning and allocation is in general inferior to an integrated approach, since an optimal partitioning can only be determined considering the optimal allocation of partitions [62].

## 8.3. Integrated Allocation Strategies

In the following we will present selected integrated allocation strategies. These consist at least of a partitioning and allocation part.

**Bubba** Copeland et al. presented the database system Bubba, which uses a partial declustering and a greedy heuristic for data placement [76]. Its partitioning approach is a simple range based declusterin that fragments each relation into a fixed number of equal-sized partitions. These partitions are allocated according to their heat (i.e. the access frequency) so that each node in the system has an equal heat. The placement strategy is a simple first-fit approach, which does not guarantee a balanced load. If the systems load is unbalanced the placement strategy is applied to the fragments with the highest temperature (i.e. heat divided by size); this way the data movement is minimized. Although Bubba was built as a parallel database system, it exploited data locality similar to our cluster database system, in contrast to other parallel database systems [197, 87] or multi disk systems [11]. However, it uses no workload-aware partitioning and uses access frequencies rather then actual costs. An overview of parallel database systems can be found in [138]. A workload aware approach was recently presented by Ozemn et al. [168]. It uses a linear programming approach to balance the workload. This approach however does not consider locality.

**Ganesan et al.** A purely dynamic approach to the allocation problem is presented by Ganesan et al. [99]. The data is range partitioned into equal sized fragments and allocated to different nodes. The approach makes no assumptions about the initial allocation. The optimization goal is storage balance, although the authors claim that the approach can be generalized to load balance. Whenever the storage on the nodes is out of balance according to a certain threshold, one of two rebalancing strategies is invoked. The *NbrAdjust* shifts weight to a node containing the neighbor fragment, while the *Reorder* strategy shares weight with an empty node. Although the algorithm aims at parallel database systems, the general idea can be adapted for reorganization in a cluster database system.

**DYFRAM** A decentralized fragmentation approach was presented by Hauglid et al. [118]. DYFRAM is similar to the approach above, a purely dynamic approach, that makes no assumptions about the initial allocation and partitioning. Based on statistics captured for each tuple, the number of local accesses versus the number of remote accesses is calculated. The following strategies are employed:

- if a fragment is often requested remotely, a copy is placed on the remote site;
- if access frequencies within a fragment vary significantly locally and remotely, the fragment is further fragmented;
- if a fragment is requested mostly from a remote site, it is moved.

The approach aims to localize the data access in a distributed system with heterogeneous access to data, which is common in federated database systems. A similar approach was presented by Stonebraker et al. [212]. This differs from our problem statement, which aims at load balancing. It is questionable whether the tuple based statistics are manageable for larger databases.

**Tashkent+** In [91] Elnikety et al. present the Tashkent+ system, a successor of the Tashkent system [90]. The main target of the Tashkent+ system is load balancing. The basic assumption is that all transactions are known in advance. Tashkent+ schedules transactions on nodes such that they can be processed in memory. Starting from a fully replicated allocation, the backends are specialized to certain transaction types. Unused relations are eventually dropped and hence an optimized data layout is generated. Due to the memory aware scheduling, a super-linear speedup is achieved. However, no partitioning is used. The approach has a similar processing model to our approach.

## 8.4. Discussion

In this chapter we have presented on overview of approaches for partitioning and allocation. We have shown several projects that use an integrated allocation strategy. In the following table the core differences of these approaches are summarized:

| Project | Partitioning | Objective |
|---|---|---|
| Bubba | Range-based | Load balancing |
| Ganesan et al. | Range-based | Storage balancing |
| DYFRAM | Tuple-based | Localization |
| Tashkent+ | No | Load balancing |

All these approaches only consider horizontal partitioning. Besides load balancing also storage balancing and localization are optimization goals. Although there is lots of work on allocation algorithms, we are not aware of an integrated approach that is based on the cluster database processing model presented in section 3.4. In the following we will present an integrated allocation strategy that allows a completely automatic allocation with horizontal and vertical partitioning and is based on the CDBS model. Goal of the allocation is load balancing.

# 9. Automatic Allocation

Partitioning and allocation traditionally are part of the design of a distributed database. Database design is the beginning of a database's life cycle. Depending on the preconditions it can be done in two forms, either top-down or bottom-up. The top-down approach is used when a database is built from scratch and the bottom-up approach is used whenever existing databases or applications have to be integrated [61]. Of course, there are practical design procedures which combine both approaches. The biggest challenge in the bottom-up approach is the integration of the preconditions. But since we will concentrate on single database applications, which feature a single integrated schema, the top-down approach is preferable. The top-down design of a database comprises several abstract levels [229]:

- Requirements analysis
- Conceptual design
- Logical design
- Physical design

In the *requirements analysis* the different goals for the use of the database are collected. These include the size of the database, user groups, applications as well as performance constraints. Based on the collected requirements a conceptual design is built. This is often done in the form of a ER-model or a UML-model [66, 9]. The resulting schema is independent of the implementation of the database management system. In the logical design phase the conceptual design is converted to the concrete schema type, which today is usually a relational schema. The logical schema is still system independent, but it is *model-dependent*. The physical design adapts the schema to the features of the chosen DBMS. This includes the selection of indexes and data allocation. A detailed overview of this database design process can be found in [89].

In distributed database systems the physical distribution of data has to be considered. Therefore a fifth level, called *distribution design* was proposed [60]. This level is located between logical and physical design. In the distribution design the global logical schema is refined by a *fragmentation schema* and an *allocation schema* [136].

However, at the design phase of a system the requirements are hard to determine and they also can change. Wrong decisions at this stage result in bad system performance. The performance at runtime strongly depends on the usage characteristics, especially in shared-nothing architectures. Since the allocation is a non trivial task, it is desirable to have an automated method that calculates an adequate allocation. As mentioned above, Daudpota defined a five step procedure for database allocation [83]:

- Collect existing global relations.
- Analyze Frequently Asked Queries.
- Set data allocation objectives.
- Transform global relations into fragment relations.
- Allocate fragment relations to sites.

Many allocation schemes follow these steps. In order to automate the allocation, all steps have to be automated. However, common allocation algorithms often contain manual procedures or prerequisites that cannot be determined exactly. Examples are the knowledge of all queries and the size of the data access of each query [169, 81]. Because of the complexity of such algorithms, it is usually not possible to alter the allocation at runtime. Therefore, we propose a simplified allocation model that can be completely automated.

Our method is also divided into the steps above. Since we aim to improve the allocation of a running system, we consider the global schema as given. To analyze frequently asked queries, we use a query log. From the query log classes of similar queries are extracted. As pointed out earlier, the main objectives of our allocation are minimal communication costs, minimizing storage costs and maximizing throughput. A further optional objective is availability and reliability. The transformation into fragmented relations is calculated by our allocation algorithms. The implementation of the allocation in the distributed system is done by a cost optimal matching. The more specific procedure is as follows:

- Query Classification
- Allocation Calculation
- Allocation Improvement (optional)
- Physical Allocation

In the first step a query history or *journal* is analyzed. In this context, queries may be read or update requests. The analysis performs a classification of the queries, queries are grouped according to the data they access. The classification determines the partitioning. If queries are grouped according to the tables they access, the allocation will have no partitioning. If the queries are grouped according to the columns they access, the allocation will probably have vertical partitioning.

Based on the classification and on the set of nodes, the allocation is calculated. Each class of similar queries or *query class* is assigned to one or more backends, so that each backend has about the same amount of work in a homogeneous system. Optionally, a meta heuristic can be used to improve the allocation for space efficiency and reduced replication.

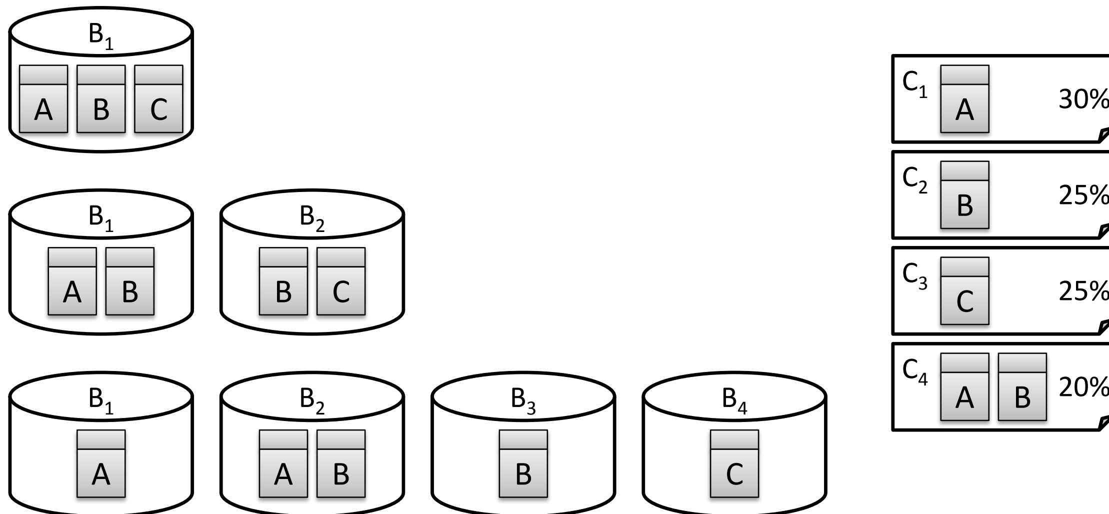



Figure 9.1.: Allocation of Read-Only Query Classes on 1 to 4 Nodes

The calculated allocation does not consider the data distribution on the backends. To keep the overhead of the reallocation as small as possible, an optimal matching of the current and the calculated allocation is computed.

Although full replication is utilized in many systems (cf. section 6.4), it usually is desirable to reduce the amount of replication for two reasons. First, replication increases disk space usage. Second, and often more important, all replicas have to be maintained. This includes update propagation as well as synchronization. Nevertheless, replication is the simplest means to increase read throughput, since read requests can be processed in parallel. Furthermore, in distributed systems the failure rate is very high; this can be compensated for by using replication.

In the following examples we will give an idea of our allocation strategy. First we will show an example for a read-only database and an optimal allocation on different cluster sizes. After that we show the more complicated read and update case.

In the first example, which can be seen in figure 9.1, a database with three relations $A$, $B$ and $C$ is allocated with partial replication, but without partitioning. The database is accessed with four types of read requests $C_1, \ldots, C_4$. The read request types are an abstraction of actual database queries. In this and the next example, queries are classified by the relations they reference. The first type of queries references relation $A$ and makes up 30% of the query workload. Accordingly, $C_2$ references relation $B$ and makes up 25% of the workload, $C_3$ references relation $C$ with 25% of the workload and $C_4$ references relations $A$ and $B$ and makes up 20% of the workload. The distributed database consists of 1 to 4 backends $B_1, \ldots, B_4$ which all have equal processing power.

Three scalings of a distributed database system can be seen. First a single database backend $B_1$: it obviously has to contain all three relations. For two backends the workload can be evenly distributed. The query types $C_1$ and $C_4$ together make up 50% of the query

load, as do $C_2$ and $C_3$. Hence, a possible solution is to allocate relations $A$ to backend $B_1$ and relation $C$ on backend $B_2$ and replicate $B$ on both backends. With this allocation both backends get an equal share of the workload and the theoretical speedup is 2. It is easy to see that this configuration is optimal in terms of space efficiency. The load distribution is shown in the following table.

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

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

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

In the first scenario, we have not considered updates. The partial replication can achieve a better theoretical speedup than a full replication for an update heavy workload, because of the reduced replication costs of updates. Again, we consider a homogeneous hardware setup. This can be seen in the second example in figure 9.2. Here, a similar setup is considered, only with an additional update workload of 20%. The exact distribution of the workload can be seen in figure 9.2. Again, three scalings of the database system are shown. For a single database there is no change to the read only scenario. For multiple backends however, the updates have to be considered. While read requests can be distributed across the nodes, where the referenced data is stored, every update request has to be executed on each replication of its referenced data. For two backends, the optimal allocation is $A$ and $B$ on backend $B_1$ and $A$ and $C$ on backend $B_2$. This results in an additional update load of 4% since $A$ is allocated twice. This can be seen in the load distribution matrix.

| | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | $U_1$ | $U_2$ | $U_3$ | Overall |
|---|---|---|---|---|---|---|---|---|
| $B_1$ | 2% | 20% | 0% | 16% | 4% | 10% | 0% | 52% |
| $B_2$ | 22% | 0% | 20% | 0% | 4% | 0% | 6% | 52% |

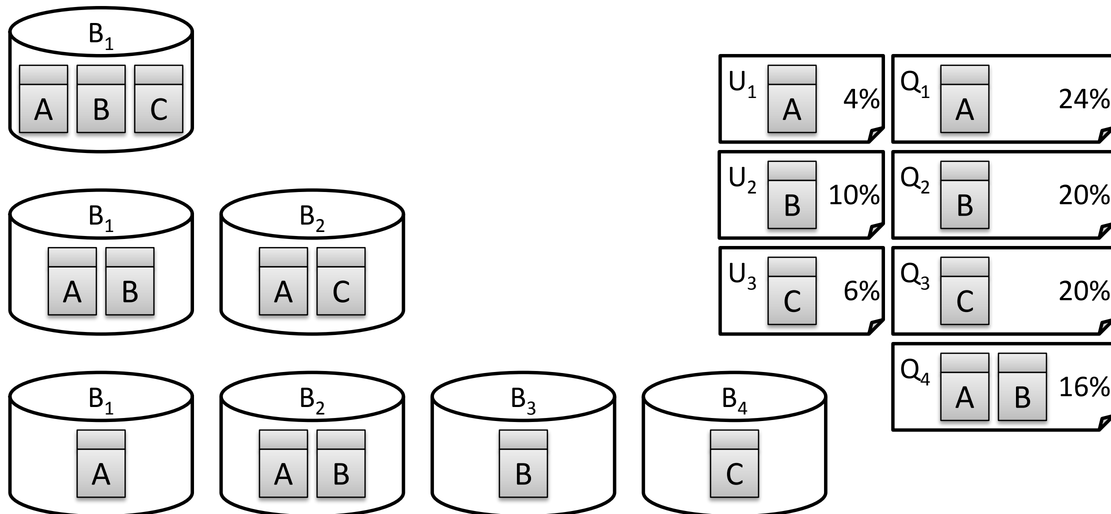



Figure 9.2.: Allocation of Read and Update Query Classes on 1 to 4 Nodes

Due to the additional update work, each of the nodes has a throughput decreased by 2%. Compared to the throughput of a single node system, the throughput is therefore decreased by 4%. To estimate the speedup the relative processing time has to be computed. The single node system uses 100% amount of time, due to the overhead each of the nodes in the two node configuration use 52% amount of time. Hence, the theoretical speedup is $\frac{100\%}{52\%} \approx 1.92$. For full replication the additional update load is 20% and the theoretical speedup 1.67. This can be seen in the load distribution table below.

| | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | $U_1$ | $U_2$ | $U_3$ | Overall |
|---|---|---|---|---|---|---|---|---|
| $B_1$ | 12% | 10% | 10% | 8% | 4% | 10% | 6% | 60% |
| $B_2$ | 12% | 10% | 10% | 8% | 4% | 10% | 6% | 60% |

For four backends the optimal allocation is not balanced any more. Backends $B_2$ and $B_3$ each get 30% of the workload, including additional updates. These two limit the maximum throughput, so each node has an overhead of 5%. Therefore, the theoretical speedup is 3.33. For a fully replicated system the theoretical speedup is 2.5.

| | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | $U_1$ | $U_2$ | $U_3$ | Overall |
|---|---|---|---|---|---|---|---|---|
| $B_1$ | 24% | 0% | 0% | 0% | 4% | 0% | 0% | 28% |
| $B_2$ | 0% | 0% | 0% | 16% | 4% | 10% | 0% | 30% |
| $B_3$ | 0% | 20% | 0% | 0% | 0% | 10% | 0% | 30% |
| $B_4$ | 0% | 0% | 20% | 0% | 0% | 0% | 6% | 26% |

This example is rather drastic, since most systems will not have a update load of 20%. Furthermore, partitioning can improve the balancing and reduce the amount of replication. Hence, in most cases better speedups can be achieved. We will give details

on the calculation of theoretical speedups considering updates in section 12. The following sections will give detailed descriptions of the single allocation steps.

## 9.1. Autonomic Allocation

Using the MAPE model presented in section 4.4, allocation can be adapted autonomically. The steps of the allocation correspond to the phases of the model:

**Monitor** In the monitoring phase information about the state of the system is gathered. For the allocation the workload is especially interesting, hence the incoming queries are logged. In addition the availability of the nodes is observed.

**Analyze** The workload is analyzed regularly. Using the query classification, the workload distribution can be calculated. If the workload distribution differs too much from the basis distribution of the allocation, it has to be adapted.

**Plan** If the current allocation is not adequate any more, a new allocation is calculated based on the current workload distribution. Using an optimal matching a cost minimal plan for the implementation of the allocation can be found.

**Execute** In order to implement the new allocation, the data has to be moved and the nodes have to be synchronized.

The resulting MAPE cycle is depicted in figure 9.3. The knowledge base for the allocation is the schema of the managed database, the number and properties of the nodes and the query history.

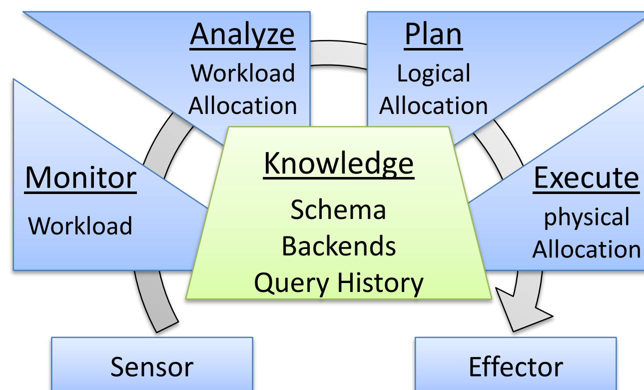



Figure 9.3.: Allocation in the MAPE Model

## 9.2. Discussion

In this section we explained the general approach for automatic allocation. We gave two examples for allocations: an allocation for systems that only have to process read accesses and an allocation for read and write access. After that we explained how the MAPE loop can be utilized to implement an automatic allocation.

In the following sections we will explain the single steps of the allocation that were defined above in detail. In section 15 we will present an evaluation of our allocation strategies our prototype uses a MAPE loop and is able to calculate the allocation completely autonomous.

# 10. Query Classification

The first step in the allocation process is the requirements analysis. For the allocation the access patterns of the data have to be determined. This can be done by examining the queries that are processed by the database system. As mentioned above, traditional approaches place the whole allocation process in the design phase of a database. Therefore, the requirements analysis has to be done on predicted usage scenarios. Obviously, for systems with large user groups and ad hoc queries this is difficult.

However, in a running system, the requirements analysis can be done based on a query profile. A query profile contains statistical information about occurring query classes and their proportional fraction of the overall query load. We will refer to the load fraction as *weight*. The query classes can be obtained from a query log. The query log usually contains the timestamp when the query was submitted and the query itself. This is not sufficient for a query profile, since there is no information about the query weight. In a homogeneous environment the execution time of a query is a good approximation of its weight. It has to be pointed out, though, that it is not completely accurate. The time that processing a certain query takes not only depends on the query itself, but to a high extent on the state of the executing system. A system under high load will need more time to execute a certain query than a system that is mostly idle. Furthermore, the state of the DBMS cache and parallel processed queries will influence the execution time. In a heterogeneous environment the execution time of a single query only has little informational value. Another way to get information about the weight of a query is to use cost analysis. This is a common database management task, since nearly all database systems use cost analysis to optimize the query plan. So another option to get the query weight is either to use additional cost estimation or, to avoid additional calculations, use the values provided by the query optimizer [172]. We will refer to a query log that contains the weight of the queries as a *journal*.

As mentioned above, the basis of an adequate allocation is to know the data access patterns. Hence, queries are classified according to the data they reference. Our methods for classification differ in the granularity of the analyzed data partitions. The most coarse classification analyzes which relation a query accesses and adds queries that access the same set of relations to the same class. A finer method uses the accessed attributes of a relation; this allows allocation of vertically fragmented relations. Analogous to this the predicates of the queries can be used in order to allow horizontal partitioning. Finally both, the accessed attributes and the predicates, can be used to supply information for hybrid fragmentations.

Consider the following example query, which is defined for a TPC-H database (see section 17.1). It calculates the total which every customer spent in the year 1995:

Listing 10.1: SQL Query to the TPC-H Database

```
select sum(O_TOTALPRICE) , C_NAME
  from ORDERS, CUSTOMER
 where O_CUSTKEY = C_CUSTKEY
   and O_ORDERDATE >= date '1995-01-01'
   and O_ORDERDATE <= date '1995-12-31'
 group by C_NAME;
```

If classified by referenced tables, the query class of the query above holds queries that reference tables ORDERS and CUSTOMER. For column-based classification the query's class holds queries that reference the following attributes:

- CUSTOMER.C_CUSTKEY
- CUSTOMER.C_NAME
- ORDERS.O_CUSTKEY
- ORDERS.O_ORDERDATE
- ORDERS.O_TOTALPRICE

Using column-based classification will result in more query classes that reference less data. In the TPC-H schema the query references only 5 of the total 17 columns of the two tables. This is especially interesting in data warehouse environments, where fact tables are frequently very broad. Finally, queries can be classified by their predicates. In the example, predicates are restrictions on O_ORDERDATE, which would result in a range partitioning of ORDERS; C_CUSTKEY = O_CUSTKEY will result in a hash partitioning of ORDERS. Using predicate-based classification will result in a large number of query classes. It is sensible to reduce the number of query classes by removing predicates on smaller tables and by grouping predicates that will result in very small partitions. However, in the allocation step related query classes will preferably be placed on the same backend. When the workload comprises a considerable amount of updates, an initial distinction of read and write queries will be performed.

Obviously, there are further forms of classification. Queries can also be classified by the originating application, or the operations they use. But for the allocation problem the classifications given above seem most promising.

## 10.1. Formal Definition

As explained above, the basis of the classification is a query journal $\mathcal{J}$. The journal is a sequence of executed queries $q$. It does not need to contain every query executed, but it should be representative[1]. The domain of $J$ is the set of all distinguishable queries $Q$,

[1] Obviously, this is a flexible definition. For the sake of simplicity, we will require a journal to contain queries of all occurring classes.

where two queries are distinguishable if they are not textually identical. Therefore, every $q \in Q$ may appear multiple times in $\mathcal{J}$. Since the order of elements is not important we will interpret $\mathcal{J}$ as a multiset with support $Q$ and characteristic function $j$. Each query $q \in Q$ accesses a set of data fragments. Depending on the classification method, a data fragment $f$ may be a relation, an attribute of a relation (or column) or a range or set of tuples (determined by predicates). Each query is classified to a single query class $C \in \mathcal{C}$, where the set of all query classes is a subset of the power set, $\mathcal{C} \subseteq \mathcal{P}(F)$ of all data fragments. The classification can be defined formally as a function:

$$\text{classify} : Q \to \mathcal{C} \tag{10.1}$$

The classify function assigns a query $q$ to a query class $C$, where $C$ is the set of data fragments referenced by $q$:

$$\text{classify}(q) = C, C = \{f \in F | q \text{ references } f\} \tag{10.2}$$

If query classes are classified column-based, they have to contain a candidate key to ensure correct reconstruction of all data. For reasons of simplicity we will presume this in the following. In the definition above reads and updates are not differentiated. For update intensive workloads, the journal is divided into update requests and read requests. The classification is analogous and the result is two sets of query classes $\mathcal{C}_U$ and $\mathcal{C}_Q$:

$$\text{classify}(q) = C, C \in \begin{cases} \mathcal{C}_Q, & \text{if } q \text{ is read} \\ \mathcal{C}_U, & \text{else} \end{cases} \tag{10.3}$$

where $\mathcal{C}_Q \cup \mathcal{C}_U = \mathcal{C}$. Strictly speaking $\mathcal{C}$ is a multiset in this case. However, for reasons of simplicity we will treat $\mathcal{C}$ like a regular set. To calculate the allocation the fraction of the overall workload produced by each query class is needed. The following function assigns this *weight* to a query:

$$\text{weight} : Q \to \mathbb{R} \tag{10.4}$$

As mentioned before, the concrete application of this function can return the execution time or a cost estimation. Using the following function the relative weight of a query class can be computed:

$$\text{weight} : \mathcal{C} \to [0, 1] \tag{10.5}$$

$$\text{weight}(C) = \frac{\sum_{q \in \{x \in Q | \text{classify}(x) = C\}} j(q) \cdot \text{weight}(q)}{\sum_{q \in Q} j(q) \cdot \text{weight}(q)} \tag{10.6}$$

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

## 10.2. Relation Based Classification

The most simple form of classification is relation based classification. It classifies queries according to the relations they reference. Since most databases comprise only a limited number of relations, only few query classes can be found. However, the classification is relatively easy. SQL queries have to name all tables used in the from clause. So the classification can be done by only parsing the from clause of a query, as could be seen in the introductory example 10.1. Consider the following query, which extracts names and phone numbers of customers with open orders served by a certain clerk.

Listing 10.2: SQL Query to the TPC-H Database

```
select C_NAME, C_PHONE
  from ORDERS, CUSTOMER
 where O_CUSTKEY = C_CUSTKEY
   and O_ORDERSTATUS = "O"
   and O_CLERK = "Clerk004420256";
```

This query retrieves a completely different result set and will have completely different runtime characteristics compared to the first example in listing 10.1. However, since the same tables are queried – ORDERS and CUSTOMER – it will be classified to the same query class. Hence, query classes will contain queries with highly different characteristics.

## 10.3. Attribute Based Classification

Attribute or column based classification considers the columns referenced by a SQL query. This includes all columns that are needed to process a query. In the example queries 10.1 and 10.2, these are the columns defined in the select and the where clause. Besides the name of the column also the name of the table is important, since relational database systems allow name conflicts on attributes in different relations. The query in listing 10.2 will be classified in a query class with the attributes:

- CUSTOMER.C_CUSTKEY
- CUSTOMER.C_NAME
- CUSTOMER.C_PHONE
- ORDERS.O_CUSTKEY
- ORDERS.O_CLERK
- ORDERS.O_ORDERSTATUS

Considering attributes will result in a finer classification. On the one hand, this will allow a more exact workload prediction. On the other hand, the classification defines possible and sensible vertical partitions. Besides partitioning, the classification information can also

be used for other schema optimizations, examples are index selection and view selection. For example, a heavy query class is an ideal candidate for a materialized view.

## 10.4. Predicate Based Classification

In large databases, queries usually only reference small amounts of the data. If a data warehouse contains several years of wholesale transactions, single queries will rarely reference the complete data set, but will instead rather reference certain data ranges. An example can be seen in the query in listing 10.1: only tuples from the ORDERS table are referenced which have values in ORDERS.O_ORDERDATE between 1995−01−01 and 1995−12−31. So the query can also be answered from a horizontal partition which only contains this data. In general, an arbitrary number of ranges on a single attribute can be found. However, it is likely that these have similar boundaries, e.g. the end of the year as above. Nonetheless, this classification will generate a huge number of query classes.

Besides ranges, comparisons can be used to classify a query. For the second query the predicates O_ORDERSTATUS = "O" and O_CLERK = "Clerk004420256" can be used for classification. This classification can then be used for hash or list partitioning, in the example the partition would only contain tuples with the according values.

Listing 10.3: SQL Query to the TPC-H Database

```
select P_NAME, count(distinct PS_SUPPKEY) as SUPPLIER_CNT
  from PART, PARTSUPP
 where P_TYPE like 'STANDARD%'
   and P_BRAND = 'BRAND#32'
   and P_PARTKEY = PS_PARTKEY
 group by P_NAME;
```

Finally, join predicates can be used for derived partitioning. Consider the query in listing 10.3. It calculates the number of suppliers of parts of a certain brand, with a name beginning with 'STANDARD'. Again the comparisons P_TYPE like 'STANDARD%' and P_BRAND = 'BRAND#32' can be used for hash or list partitioning. The tables PART and PARTSUPP are joined with the condition P_PARTKEY = PS_PARTKEY. Using derived partitioning, the table PARTSUPP can be hash partitioned using the predicate P_BRAND = 'BRAND#32'.

## 10.5. Hybrid Classification

Obviously, it is possible to classify queries according to their referenced columns and their predicates. The result is an even finer classification that can be used for horizontal and vertical partitioning. To be consistent with the classification formulation above, we use vertical and horizontal partitioning equally, leading to an equal partitioning.

## 10.6. Discussion

In this section we have presented our approach to query classification. It is the basis of the allocation algorithms presented in the next sections. The query classification can be done in various granularities. We have shown examples for relation based, attribute based, and predicate based classification. These can be combined to a form of hybrid classification.

All presented classifications build an implicit partitioning of the data and hence determine the granularity of the distribution units in the allocation. While table based classification enables a distribution of complete tables, attribute based classification potentially enables a vertical partitioning, finally predicate based classification enables horizontal partitioning. The more fine grained the classification is, the better the performance of the allocation can be since the load can be distributed more evenly. However, a more fractured allocation also increases the processing overhead to some extent. Nevertheless, this overhead does not exceed the gains in the performance. This will be shown in our test results in section 15.

# 11. Allocation – Read Mostly

With a given query classification the allocation problem is how to assign query classes to database backends, such that all backends have a load fitting their capabilities and that the disk usage is minimized. We will first give a formal definition of the problem, then prove that it is NP-hard and after that show different approaches to calculate solutions.

## 11.1. Formal Definition

The allocation can be formally described as a function that assigns data fragments $f \in F$ to the backend databases $B \in \mathcal{B}$. Since we aim for an balanced load we distinguish backends by their query processing performance. This measure is given in relation to the sum of the query processing performances of all backends. Thus, the domain of the relative performance is $[0, 1]$. As the basis of our allocation is a classification, input of the allocation are the query classes $C \in \mathcal{C}$. The result is a partial replication. It can be represented as a multiset of backends with domain $\mathcal{P}(F) \times [0, 1]$, $\mathcal{B} = \langle \mathcal{P}(F) \times [0, 1], b \rangle$, where $b$ is the characteristic function of $\mathcal{B}$. A backend is therefore a pair $\langle B^*, l \rangle$ of a set of fragments $B^* \subseteq F$ and a maximum relative load $l \in [0, 1]$. Hence, the allocation is defined as:

$$\text{allocation} : \mathcal{P}(\mathcal{P}(F)) \rightarrow \mathcal{P}^{\mathcal{P}(F)\times[0,1]} \tag{11.1}$$

$$\text{allocation}(\mathcal{C}) = \mathcal{B},\ \forall C \in \mathcal{C}, \exists B \in \mathcal{B}, B = \langle B^*, l \rangle : f \in C \Rightarrow f \in B^* \tag{11.2}$$

The definition assures that every query class is assigned to at least one backend. In a homogeneous environment all backends will get the same share of the overall workload. In a heterogeneous environment the shares of the workload that backends can handle may differ. To simplify the following equations, we introduce the function load which extracts the relative performance of a backend:

$$\text{load} : \mathcal{B} \rightarrow [0, 1] \tag{11.3}$$

$$\text{load}(\langle B^*, l \rangle) = l \tag{11.4}$$

$$\sum_{B \in \mathcal{B}} \text{load(B)} = 1 \tag{11.5}$$

In a homogeneous cluster of $s$ nodes the relative load is $\frac{1}{s}$. The set of fragments of a backend can extracted by the following function:

$$\text{fragments} : \mathcal{B} \rightarrow F \tag{11.6}$$

$$\text{fragments}(\langle B^*, l\rangle) = B^* \tag{11.7}$$

To express the part of the weight of a query that is assigned to a backend the function assign is used:

$$\text{assign} : \mathcal{C} \times \mathcal{B} \to [0, 1] \tag{11.8}$$

$$\text{assign}(C, B) > 0 \Rightarrow C \subseteq \text{fragments}(B) \tag{11.9}$$

An allocation is valid, i.e. the workload is balanced and all query classes are allocated if the following constraints are satisfied:

$$\sum_{B \in \mathcal{B}} \text{assign}(C, B) = \text{weight}(C) \tag{11.10}$$

$$\sum_{C \in \mathcal{C}} \text{assign}(C, B) = \text{load}(B) \tag{11.11}$$

Constraints (11.10) and (11.11) result in the following equation:

$$\sum_{B \in \mathcal{B}} \sum_{C \in \mathcal{C}} \text{assign}(C, B) = 1 \tag{11.12}$$

An allocation satisfying these constraints will enable the scheduler to balance the load on the backends, nevertheless it is in general not minimal. This means that such an allocation may contain more replicated fragments than needed. To get an allocation with minimal replication the following sum has to be minimized:

$$\sum_{B \in \mathcal{B}} \sum_{f \in \text{fragments}(B)} 1 \tag{11.13}$$

The sum calculates the number of allocated fragments in the allocation. Since the data fragments usually have different sizes, it is desirable to minimize disk usage. For this the size of the data fragments is needed. We use the function size that maps a fragment to its size:

$$\text{size} : F \to \mathbb{R} \tag{11.14}$$

To get the minimal allocation in terms of disk usage we have to minimize the following sum:

$$\sum_{B \in \mathcal{B}} \sum_{f \in \text{fragments}(B)} \text{size}(f) \tag{11.15}$$

To ensure completeness, data fragments $f \in F$ that are not already allocated have to be assigned to a backend. Basically, they can be assigned to any backend, in the sense of minimality as defined above and according to the theoretical throughput there is no difference. To ensure a less fractured schema, it is sensible to place the fragments on backends where related data is allocated.

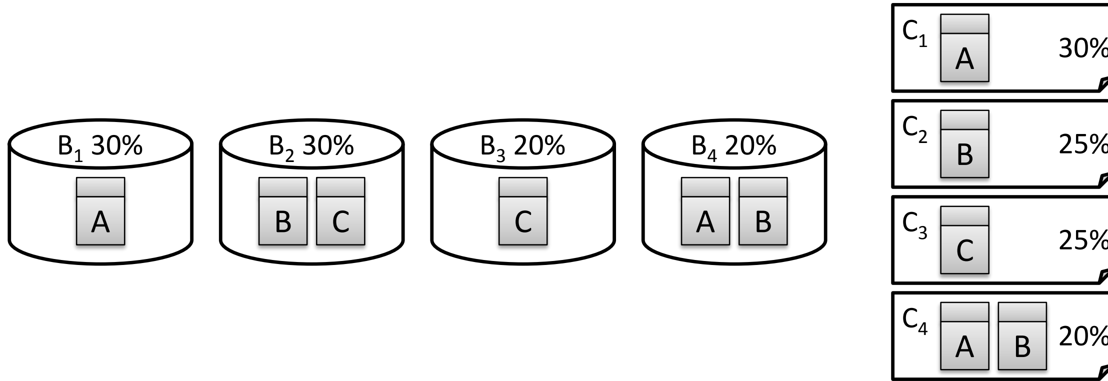



Figure 11.1.: Allocation of Read-Only Query Classes on Heterogeneous Backends

## 11.2. Optimal Allocation

The allocation can be described by a matrix $\mathfrak{A} \in \{0,1\}^{|\mathcal{B}| \times |F|}$, which contains a row for each backend and a column for each fragment. The allocation matrix for the example in figure 11.1 is:

$$\mathfrak{A} = \begin{array}{c} \\ B_1 \\ B_2 \\ B_3 \\ B_4 \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \end{array} \tag{11.16}$$

In general the allocation matrix is defined as:

$$\mathfrak{A} : \begin{cases} \{1,..,|\mathcal{B}|\} \times \{1,..,|F|\} \to \{0,1\} \\ \qquad\qquad (i,j) \mapsto a_{ij} \end{cases} \tag{11.17}$$

Additionally, a load distribution matrix $\mathfrak{L} \in [0,1]^{|\mathcal{B}| \times |\mathcal{C}|}$ is defined which contains the partial load of each query class per backend. For the example a balanced load is:

$$\mathfrak{L} = \begin{array}{c} \\ B_1 \\ B_2 \\ B_3 \\ B_4 \end{array} \begin{array}{c} \begin{array}{cccc} C_1 & C_2 & C_3 & C_4 \end{array} \\ \begin{pmatrix} 0.30 & 0 & 0 & 0 \\ 0 & 0.25 & 0.05 & 0 \\ 0 & 0 & 0.20 & 0 \\ 0 & 0 & 0 & 0.20 \end{pmatrix} \end{array} \tag{11.18}$$

The general definition is:

$$\mathfrak{L} : \begin{cases} \{1,..,|\mathcal{B}|\} \times \{1,..,|\mathcal{C}|\} \to [0,1] \\ \qquad\qquad (i,k) \mapsto l_{ik} \end{cases} \tag{11.19}$$

With this matrix it is easy to see that for a given set of backends $\mathcal{B}$ and a set of query classes $\mathcal{C}$, there are $2^{|\mathcal{B}|\times|\mathcal{C}|}$ possible allocations. Because of the exponential number of allocations exhaustive search is not an option to compute an optimal allocation. However, the allocation problem can be formulated as a mixed integer linear program. Using the matrix representation, the allocation problem can be characterized as a minimization problem analogous to the definition in equation 11.15:

$$\min \sum_{a_{ij}\in\mathfrak{A}} a_{ij} \cdot \text{size}(f_j) \tag{11.20}$$

Again equations 11.10 and 11.11 must be satisfied. These can be formulated using the matrices as follows:

$$\forall B_i \in \mathcal{B}, l_{ik} \in \mathfrak{L} : \sum_{k=1}^{|\mathcal{C}|} l_{ik} = \text{load}(B_i) \tag{11.21}$$

$$\forall C_k \in \mathcal{C}, l_{ik} \in \mathfrak{L} : \sum_{i=1}^{|\mathcal{B}|} l_{ik} = \text{weight}(C_k) \tag{11.22}$$

To establish the connection between the query classes and their referenced data fragments, additional helper variables $h_{ik} \in \{0, 1\}$ are needed:

$$\mathfrak{H} : \begin{cases} \{1, .., |\mathcal{B}|\} \times \{1, .., |\mathcal{C}|\} \to \{0, 1\} \\ \hfill (i, k) \mapsto h_{ik} \end{cases}, \quad h_{ik} = \begin{cases} 1, \text{ if } l_{ik} > 0 \\ 0, \text{ else} \end{cases} \tag{11.23}$$

If $h_{ik} = 1$, then query class $C_k$ is allocated to backend $B_i$. Therefore, all fragments $f_j$ which are referenced by query class $C_k$ have to be allocated to backend $B_i$. This can be expressed by the following constraint:

$$\forall C_k \in \mathcal{C}, \forall B_i \in \mathcal{B} : \sum_{j:f_j\in C_k} a_{ij} \geq |C_k| * h_{ik} \tag{11.24}$$

This constraint sums up the number of fragments of query class $C_k$, which are allocated to backend $B_i$. If $C_k$ is allocated to $B_i$ it must be equal to the number of fragments in $C_k$, otherwise it can be greater than 0. With these definitions a linear program can be specified. For the example above this can be seen in listing 11.1. The source is written in lp_solve[1] syntax. To improve readability, unnecessary constraints and constants were omitted. The source starts with the objective function; this is the same as the definition in equation 11.20. Since all tables (i.e. data fragments) have the same weight, the weighting factors are omitted. Then, the restrictions of the allocation are introduced, i.e. the weight of the query classes and the maximum load of the backends. These are the row and column sums of the load distribution matrix $\mathfrak{L}$. After that, the helper variables are

[1] lp_solve is an open source linear program solver, it is available at `http://lpsolve.sourceforge.net/` (last visited 2011-04-15)

Listing 11.1: Linear Program for an Optimal Allocation

```
/* Minimization problem */
min: a_11 + a_12 + a_13
   + a_21 + a_22 + a_23
   + a_31 + a_32 + a_33
   + a_41 + a_42 + a_43;

/* Allocation of the query */
/* classes                */
l_11 + l_21 + l_31 + l_41 = 0.3;
l_12 + l_22 + l_32 + l_42 = 0.25;
l_13 + l_23 + l_33 + l_43 = 0.25;
l_14 + l_24 + l_34 + l_44 = 0.2;

/* Load constraints for backends */
l_11 + l_12 + l_13 + l_14 = 0.3;
l_21 + l_22 + l_23 + l_24 = 0.3;
l_31 + l_32 + l_33 + l_34 = 0.2;
l_41 + l_42 + l_43 + l_44 = 0.2;

/* If query class k is on */
/* backend i it has to be */
/* allocated              */
h_11 - l_11 >= 0;
h_12 - l_12 >= 0;
h_13 - l_13 >= 0;
h_14 - l_14 >= 0;

h_21 - l_21 >= 0;
h_22 - l_22 >= 0;
h_23 - l_23 >= 0;
h_24 - l_24 >= 0;

h_31 - l_31 >= 0;
h_32 - l_32 >= 0;
h_33 - l_33 >= 0;
h_34 - l_34 >= 0;
h_41 - l_41 >= 0;
h_42 - l_42 >= 0;
h_43 - l_43 >= 0;
h_44 - l_44 >= 0;

/* All fragments j of query */
/* class k have to be       */
/* allocated on backend i   */
a_11 - h_11 >= 0;
a_21 - h_21 >= 0;
a_31 - h_31 >= 0;
a_41 - h_41 >= 0;

a_12 - h_12 >= 0;
a_22 - h_22 >= 0;
a_32 - h_32 >= 0;
a_42 - h_42 >= 0;

a_13 - h_13 >= 0;
a_23 - h_23 >= 0;
a_33 - h_33 >= 0;
a_43 - h_43 >= 0;

a_11 + a_12 - 2 h_14 >= 0;
a_21 + a_22 - 2 h_24 >= 0;
a_31 + a_32 - 2 h_34 >= 0;
a_41 + a_42 - 2 h_44 >= 0;

/* a_ij and h_ik are binary */
bin a_11, a_12, a_13,
    a_21, a_22, a_23,
    a_31, a_32, a_33,
    a_41, a_42, a_43,
    h_11, h_12, h_13, h_14,
    h_21, h_22, h_23, h_24,
    h_31, h_32, h_33, h_34,
    h_41, h_42, h_43, h_44;
```

introduced, according to equation 11.23. For each query class all relevant fragments have to be allocated, as defined in equation 11.24. Finally, all $a_{ij}$ and $h_{ik}$ are declared as binary variables. It would also be possible to introduce boundaries for $l_{ik}$, but since all variables are non-negative by default this is not necessary.

The number of variables in the linear program is $|\mathfrak{A}| + |\mathfrak{L}| + |\mathfrak{H}|$ and the number of constraints $|\mathcal{C}| + |\mathcal{B}| + 2 \cdot |\mathfrak{H}|$. Since mixed integer linear programming is NP-complete, only small problem instances are solvable using this approach.

## 11.3. NP-Hardness of the Allocation

In this section, we will show that the allocation problem as defined above is NP-hard. The general approach to proving NP-hardness is finding a suitable NP-complete problem and reducing it to the problem at hand. The reduction has to have polynomial time complexity. For our problem a reduction can be found from the strong NP-complete problem 3-Partition. Strong NP-completeness indicates that a problem is hard to solve, even for relatively small instances. The following definition is roughly taken from [101]:

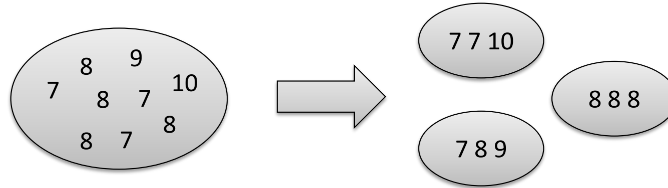



Figure 11.2.: Example for the 3-Partition Problem.

**3-Partition** The 3-partition problem decides if a multiset of integers can be divided into subsets of size 3, such that the sum of the elements of each subset is equal. Or more formally, given a finite multiset $A$ of $3m$ elements, a bound $b \in \mathbb{N}^+$ and a size size : $A \to \mathbb{N}^+$. Let the size of each element be such that $b/4 < \text{size}(a) < b/2$ and $\sum_{a \in A} \text{size}(a) = mb$. Then the question is if $A$ can be partitioned into $m$ disjoint multisets $S_1, S_2, \ldots, S_m$ such that for all $S_i$, with $1 \leq i \leq m$, the constraint $\sum_{a \in S_i} \text{size}(a) = b$ is satisfied.

Consider the example in figure 11.2, given a multiset $A = \{7, 7, 7, 8, 8, 8, 8, 9, 10\}$, with $|A| = 9$ and $\sum_{a \in A} = 72$. Since $A$ has 9 elements, $m = 3$ and $b = \sum_{a \in A} /m = 24$. All elements $a \in A$ lie within the required bounds $6 < a < 12$. The 3-partition problem decides if $A$ can be partitioned into 3 multisets, each has the sum 24, which is possible for the example.

*Proof.* To reduce 3-partition to the allocation problem, the following configuration can be considered: let $\mathcal{C}$ be a set of $3m$ disjoint query classes, where each query class $C \in \mathcal{C}$ has exactly one element $f$ with $\text{size}(f) = 1$. Each query class has a weight $b/4 < \text{weight}(C) <$

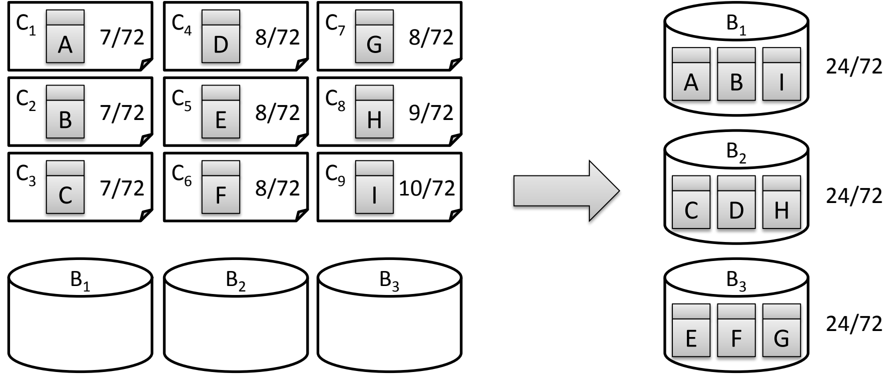



Figure 11.3.: Cluster Allocation Solution to the 3-Partition Problem

$b/2$ and the overall weight is $\sum_{C \in \mathcal{C}} \text{weight}(C) = mb$. This differs from the previous definition; however, it is stricter and can be translated by normalizing the weight of all query classes to 1. There are $m$ homogeneous backends. Let $\text{allocation}(\mathcal{C}) = \mathcal{B}$ be a minimal allocation. Each backend has equal load of $\text{load}(B) = b$. The idea behind the proof can be seen in figure 11.3.

If an optimal allocation $\mathcal{B}$ has the property: $\sum_{B \in \mathcal{B}} |\text{fragments}(B)| = 3m$, then $\forall B \in \mathcal{B} : |\text{fragments}(B)| = 3$ and each $C \in \mathcal{C}$ is assigned to exactly one backend. Since the weight of each query class $C$ is $b/4 < \text{weight}(C) < b/2$, each backend has to contain more than 2 query classes. If no query class has to be replicated, each backend contains 3 query classes. Then $\mathcal{C}$ can be partitioned in $m$ disjoint sets $S_1, \ldots, S_m$, with $S_i = \{C \in \mathcal{C} | \text{assign}(C, B_i) > 0\}$, such that $\sum_{C \in S_i} \text{weight}(C) = b$.

Such an allocation can only exist if the set of weights of the query classes can be partitioned into $m$ sets with an equal sum. Hence, the allocation algorithm generates a solution for the 3-partition problem. $\square$

## 11.4. Greedy Heuristic

The allocation problem is similar to bin packing. Therefore, we use a first-fit strategy to calculate the allocation in polynomial time [72]. Input of the algorithm is the classification and an empty set of backends; output is the set of backends with allocated query classes.

The heuristic starts by sorting the query classes descending by their weight multiplied by the size of the referenced fragments in line 1. This ensures that query classes that require the most space will be allocated first. We define variables freeCapacity and restWeight that store the remaining processing capacity for each backend and the rest of the workload of each query class.

Then, for each query class the difference to the query classes already allocated on each backend is calculated in the inner foreach loop starting in line 5. If the load of the backend is already reached, the difference is $\infty$. If no query class is allocated to a backend, the

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

1 $\mathfrak{C} \leftarrow$ sort $\mathcal{C}$ descending to weight$(C) \times$ size$(C)$;
2 freeCapacity$(\mathcal{B}) \leftarrow$ load$(\mathcal{B})$;
3 restWeight$(\mathfrak{C}) \leftarrow$ weight$(\mathfrak{C})$;
4 **foreach** $C \in \mathfrak{C}$ **do**
5     **foreach** $B \in \mathcal{B}$ **do**
6         **if** *freeCapacity*$(B) = 0$ **then**
7             difference$(C, B) \leftarrow \infty$;
8         **else if** *freeCapacity*$(B) =$ *load*$(B)$ **then**
9             difference$(C, B) \leftarrow 0$;
10         **else**
11             difference$(C, B) \leftarrow$ size$(C \setminus$ fragments$(B))$;
12         **end**
13     **end**
14     **while** *restWeight*$(C) > 0$ **do**
15         $B \leftarrow B \in \mathcal{B}$ with difference$(C, B)$ minimal;
16         fragments$(B) \leftarrow$ fragments$(B) \cup C$;
17         **if** *restWeight*$(C) \leq$ *freeCapacity*$(B)$ **then**
18             assign$(C, B) \leftarrow$ restWeight$(C)$;
19             freeCapacity$(B) \leftarrow$ freeCapacity$(B) -$ restWeight$(C)$;
20             restWeight$(C) \leftarrow 0$;
21         **else**
22             assign$(C, B) \leftarrow$ restWeight$(C) -$ freeCapacity$(B)$;
23             restWeight$(C) \leftarrow$ restWeight$(C) -$ assign$(C, B)$;
24             freeCapacity$(B) \leftarrow 0$;
25         **end**
26         difference$(C, B) \leftarrow \infty$;
27     **end**
28 **end**
29 **return** $\mathcal{B}$

**Algorithm 11.1:** Greedy Allocation Algorithm

difference is 0; or it is the size of the fragments that have to be additionally allocated.

The query class is assigned to the backend with the least difference in the while loop starting in line 14. Two cases have to be distinguished: either the remaining workload of the query class can be completely assigned to the backend; then the free capacity of the backend is updated and the next query class can be processed. If the workload of query class cannot be completely processed by the best fitting backend, then as much weight of the query class as possible is placed on the backend and the rest weight of the query class

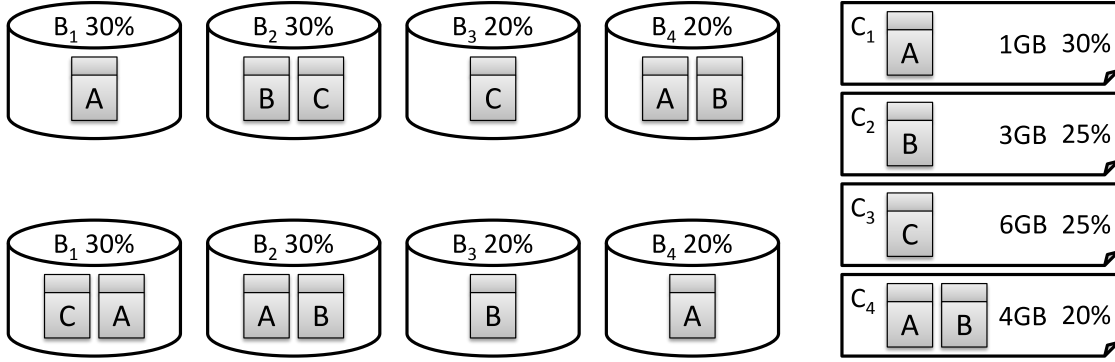



Figure 11.4.: Allocation of Different-Sized Tables on Heterogeneous Backends with (below) and without (above) Consideration of the Size

is updated. This is repeated until all query classes are assigned.

Since the outer loop is repeated $|\mathcal{C}|$ times and the inner while loop at most $|\mathcal{B}|$ times, the runtime of the algorithm lies in $O(|\mathcal{C}| \cdot |\mathcal{B}|)$.

Consider the example in figure 11.4. An allocation for 4 query classes with different-sized relations to 4 backends is shown. In the upper allocation is optimal in terms of number of replicas, while the lower allocation is also optimal in terms of required space. Based on the lower example we will explain the function of the greedy algorithm.

Input of the allocation algorithm is the set of query classes $\mathcal{C} = C_1, C_2, C_3, C_4$ and the four empty backends $B_1, \ldots, B_4$. Initially, the query classes are sorted according to the product of their size and relative weight (line 1). For $C_3$ the product is $6 \cdot 0.25 = 1.5$ and for $C_4$ $(1 + 3) \cdot 0.2 = 0.8$. Result is the following descending list or sequence of the query classes: $\mathfrak{C} = (C_3, C_4, C_2, C_1)$. Then a list of the capacities of the backends is generated, for the empty backends this is equal to their relative processing power:

$$\text{freeCapacity} = (B_1 : 0.3, B_2 : 0.3, B_3 : 0.2, B_4 : 0.2) \tag{11.25}$$

An other auxiliary list contains the weight of each query class that has not yet been allocated:

$$\text{restWeight} = (C_1 : 0.3, C_2 : 0.25, C_3 : 0.25, C_4 : 0.2) \tag{11.26}$$

Then the main loop iterates over the sorted set of query classes (line 4). To find the most suitable backend, the fragments that are referenced by the query class are compared with the fragments that are already allocated to the backends (line 5).

For query class $C_3$ the computed difference is 0 for all backends, since all backends are empty. The query class is allocated to the first backend with minimal difference, so in the given case $B_1$. The backend now contains the fragment $C$, and the restWeight and freeCapacity are updated. Since the query class can be fully allocated to backend $B_1$ the new restWeight is: restWeight $= (C_1 : 0.3, C_2 : 0.25, C_3 : 0, C_4 : 0.2)$. The new

freeCapacity is: freeCapacity $= (B_1 : 0.05, B_2 : 0.3, B_3 : 0.2, B_4 : 0.2)$. The allocation matrix is now:

| | $A$ | $B$ | $C$ |
|---|---|---|---|
| $B_1$ | 0 | 0 | 1 |
| $B_2$ | 0 | 0 | 0 |
| $B_3$ | 0 | 0 | 0 |
| $B_4$ | 0 | 0 | 0 |

The load matrix is:

| | $C_1$ | $C_2$ | $C_3$ | $C_4$ | Overall |
|---|---|---|---|---|---|
| $B_1$ | 0% | 0% | 25% | 0% | 25% |
| $B_2$ | 0% | 0% | 0% | 0% | 0% |
| $B_3$ | 0% | 0% | 0% | 0% | 0% |
| $B_4$ | 0% | 0% | 0% | 0% | 0% |

Since $C_1$ was fully allocated, the next query class is processed. For $C_4$ again the differences are computed, for $B_1$ the difference is 4, for the other backends it is 0. Hence, the query class is allocated on backend $B_2$ and restWeight and freeCapacity are updated: restWeight $= (C_1 : 0.3, C_2 : 0.25, C_3 : 0, C_4 : 0)$, freeCapacity $= (B_1 : 0.05, B_2 : 0.1, B_3 : 0.2, B_4 : 0.2)$. The allocation matrix is now:

| | $A$ | $B$ | $C$ |
|---|---|---|---|
| $B_1$ | 0 | 0 | 1 |
| $B_2$ | 1 | 1 | 0 |
| $B_3$ | 0 | 0 | 0 |
| $B_4$ | 0 | 0 | 0 |

The load matrix is:

| | $C_1$ | $C_2$ | $C_3$ | $C_4$ | Overall |
|---|---|---|---|---|---|
| $B_1$ | 0% | 0% | 25% | 0% | 25% |
| $B_2$ | 0% | 0% | 0% | 20% | 20% |
| $B_3$ | 0% | 0% | 0% | 0% | 0% |
| $B_4$ | 0% | 0% | 0% | 0% | 0% |

Again, $C_4$ is fully allocated and the next query class is processed. The differences for $C_2$ are: $(C_2, B_1) : 3, (C_2, B_2) : 0, (C_2, B_3) : 0, (C_2, B_4) : 0$. The query class $C_2$ gets allocated to the first backend with minimal difference $B_2$ and restWeight and freeCapacity are updated: restWeight $= (C_1 : 0.3, C_2 : 0.15, C_3 : 0, C_4 : 0)$, freeCapacity $= (B_1 : 0.05, B_2 : 0, B_3 : 0.2, B_4 : 0.2)$. Since the $C_2$ is not fully allocated, it has to be allocated to another backend. The difference to $B_2$ is updated to $\infty$. The second backend with minimal difference is $B_3$. Again, restWeight and freeCapacity are updated: restWeight $= (C_1 : 0.3, C_2 : 0, C_3 : 0, C_4 : 0)$, freeCapacity $= (B_1 : 0.05, B_2 : 0, B_3 : 0.05, B_4 : 0.2)$. $C_2$ is now fully allocated and the last query class $C_1$ can be processed. The allocation matrix is:

| | $A$ | $B$ | $C$ |
|---|---|---|---|
| $B_1$ | 0 | 0 | 1 |
| $B_2$ | 1 | 1 | 0 |
| $B_3$ | 0 | 1 | 0 |
| $B_4$ | 0 | 0 | 0 |

The load matrix is:

| | $C_1$ | $C_2$ | $C_3$ | $C_4$ | Overall |
|---|---|---|---|---|---|
| $B_1$ | 0% | 0% | 25% | 0% | 25% |
| $B_2$ | 0% | 10% | 0% | 20% | 30% |
| $B_3$ | 0% | 15% | 0% | 0% | 15% |
| $B_4$ | 0% | 0% | 0% | 0% | 0% |

The differences for $C_1$ are: $(C_1, B_1) : 1, (C_1, B_2) : \infty, (C_1, B_3) : 1, (C_1, B_4) : 0$. Therefore $C_1$ allocated $B_4$ and restWeight and freeCapacity are updated: restWeight $= (C_1 : 0.2, C_2 : 0, C_3 : 0.0, C_4 : 0)$, freeCapacity $= (B_1 : 0.05, B_2 : 0, B_3 : 0.05, B_4 : 0)$. The difference to $B_4$ is updated to $\infty$. $C_1$ is then allocated to $B_1$, the updates are: restWeight $= (C_1 : 0.05, C_2 : 0, C_3 : 0.0, C_4 : 0)$, freeCapacity $= (B_1 : 0, B_2 : 0, B_3 : 0.05, B_4 : 0)$. Finally, the rest of $C_2$ is allocated to $B_2$. Since all query classes are completely allocated the algorithm stops. The allocation matrix is:

| | $A$ | $B$ | $C$ |
|---|---|---|---|
| $B_1$ | 1 | 0 | 1 |
| $B_2$ | 1 | 1 | 0 |
| $B_3$ | 1 | 1 | 0 |
| $B_4$ | 1 | 0 | 0 |

The final load matrix is:

| | $C_1$ | $C_2$ | $C_3$ | $C_4$ | Overall |
|---|---|---|---|---|---|
| $B_1$ | 5% | 0% | 25% | 0% | 30% |
| $B_2$ | 0% | 10% | 0% | 20% | 30% |
| $B_3$ | 5% | 15% | 0% | 0% | 20% |
| $B_4$ | 20% | 0% | 0% | 0% | 20% |

As can be seen in relation to the optimal allocation in figure 11.4 this allocation is not optimal. In general, the heuristic calculates good, but not optimal results in most cases (this can also be seen in the evaluation in section 15.1). Consider the example in figure 11.5. Here, a database with three tables $A$, $B$, and $C$ of equal size is allocated on three backends with equal processing power. There are four query classes, where query class $C_1$ references $A$, $C_2$ references $B$, $C_3$ references $C$, and $C_4$ references $A$, $B$ and $C$. The first three have an equal share of 31% of the workload and query class $C_4$ has only 7%. Due to their larger workload, first query classes $C_1$, $C_2$, and $C_3$ are allocated. Obviously, the

heuristic will place them on separate backends, since they reference disjoint data. When $C_4$ is allocated, it has to be placed on all backends, resulting in a full replication. It is easy to see that this is not an optimal solution, as can be seen in figure 11.5. However, the results of the greedy heuristic can be further optimized using meta heuristics.

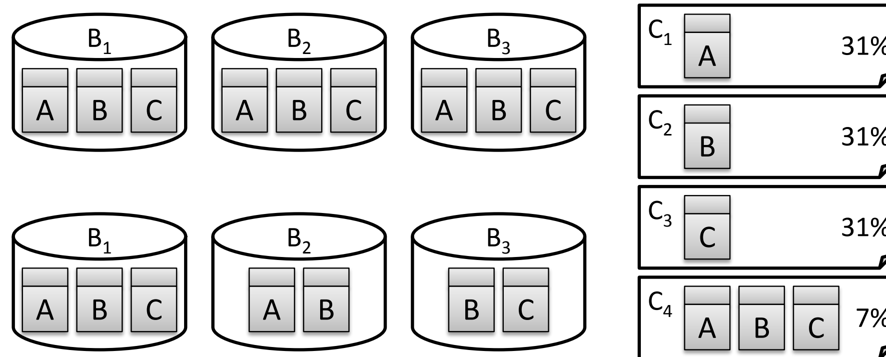



Figure 11.5.: Heuristic (above) vs. Optimal (below) Allocation

## 11.5. Meta Heuristics

In general, the greedy heuristic computes a valid, but not an optimal solution. By altering the heuristically found solution, it is possible to generate a better solution. Since the search space is exponential, not all possible solutions can be tested. A common approach is to generate valid mutations of the initial solution randomly. For the allocation, it is possible to alter the allocation matrix $\mathfrak{A}$, while ensuring the validity of the allocation. To control the number and order of the mutations a meta heuristic can be used. Many meta heuristics have a similar approach:

**Initialization** An initial solution is generated randomly or deterministically.

**Mutation** A number of mutations is generated randomly.

**Evaluation** The mutations are evaluated according to a cost function.

**Selection** A new initial solution is selected from the mutations and the initial solution.

**Termination** After a predetermined number of iterations or a defined stopping condition the best solution is returned.

The algorithms differ mostly in the way in which the mutations are selected. An overview of different approaches can be found in [222]. For the allocation problem an evolutionary programming approach was chosen. Evolutionary programming differs from other evolutionary approaches, such as genetic algorithms and evolution strategies, since it uses no recombination of solutions [39]. Since we also use local improvements the algorithm can be classified as hybrid heuristic or memetic algorithm [158].

### 11.5.1. Evolutionary Algorithm

Evolutionary algorithms use a set of solutions that store the current population. For each iteration a new population $\mathfrak{P}$ with a determined number of old solutions and mutations is generated. After a fixed number of iterations or a stopping condition the best solution is returned. A pseudo code formulation can be found in algorithm 11.2.

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

1 $\mathfrak{P} \leftarrow \{S_{init}\}$ ;
2 **for** *number of iterations* **do**
3 | $\mathfrak{P}' \leftarrow \text{mutate}(\mathfrak{P}, p)$;
4 | $\mathfrak{P} \leftarrow \text{select}(\mathfrak{P}, \frac{2}{3}, \text{best}) \cup \text{select}(\mathfrak{P}', \frac{1}{3}, \text{best})$;
5 | $\mathfrak{I} \leftarrow \text{select}(\mathfrak{P}, \frac{1}{3}, \text{random})$;
6 | $\mathfrak{P} \leftarrow \mathfrak{P} \setminus \mathfrak{I}$;
7 | **foreach** $S \in \mathfrak{I}$ **do**
8 | | $S \leftarrow \text{improve}(S)$;
9 | **end**
10 | $\mathfrak{P} \leftarrow \mathfrak{P} \cup \mathfrak{I}$;
11 **end**
12 $\mathcal{S}_{min} \leftarrow \{S \in \mathfrak{P} | \text{weight}(S) = \min_{S' \in \mathfrak{P}} \text{weight}(S')\}$;
13 **return** $\mathcal{S}_{min}$

**Algorithm 11.2:** Evolutionary strategy

The algorithm starts with the generation of the initial population in line 2. In general, this can be a set of randomly generated allocations or simply a full replication. For a faster convergence of the algorithm, we start with the solution of the greedy heuristic. In the loop, in line 1, the evolutionary process is executed, the number of iterations determines the runtime. Another common stopping criterion is to stop if for a certain number of iterations no better solutions have been found. However, this makes the runtime of the algorithm non-deterministic. The first step in the evolutionary approach is to mutate the population to generate the offspring (line 2). In this step a new set of allocations is generated by randomly altering the current population. As is common in evolutionary programming the mutation is based on single parents instead of combining parents. The exact procedure is described below in section 11.5.2. After that, the new population is chosen (line 3). The strategy is a so-called $(\lambda + \mu)$ approach. This means that parents and offspring are mixed for the new population, instead of only using the offspring. The function select$(X, y, \Theta)$ choses a fraction $y$ of the set $X$ using the operator $\Theta$. For the new population, the best $\frac{2}{3}$ of the old population and the best $\frac{1}{3}$ of the offspring survive. In contrast to a classic evolutionary program, the memetic algorithm now chooses randomly $\frac{1}{3}$ of the new population that is improved using a local search (lines 4 - 9). The local search methods can be found below in section 11.5.3. After a certain number of iterations

the best solutions, i.e. all solutions with minimal costs, are returned.

### 11.5.2. Mutation

To generate mutations, existing solutions are altered randomly to generate new solutions. However, the generated allocations have to be valid in terms of the constraints defined in section 11.1. As stated above, the allocation problem is similar to the bin packing problem. Hence, a similar mutation strategy can be applied. Alvim et al. presented an improvement strategy for bin packing, where two bins are chosen and the items contained are redistributed to the bins [21]. This can also be applied to the allocation problem. From an initial solution, two backends are randomly chosen and the allocated query classes are redistributed to the backends. To generate random mutations the order in which the query classes are allocated can be randomized. Consider the example in figure 11.5. In the non-optimal solution, backends $B_1$ and $B_2$ have the following allocation:

| $B_1:$ | 33.3% | $B_2:$ | 33.3% |
|---|---|---|---|
| $C_1:$ | 31.0% | $C_2:$ | 31.0% |
| $C_4:$ | 2.3% | $C_4:$ | 2.3% |

To redistribute the query classes, first the set of all query classes that are allocated to the two backends is generated. It is $\{C_1 : 31\%, C_2 : 31\%, C_4 : 6.6\%\}$. This set is shuffled and the query classes are allocated to the backends. For example, if the new order of query classes is $C_2, C_4, C_1$, then first query class $C_2$ will be allocated, then query class $C_4$ and finally query class $C_1$. If multiple backends can hold the query class it will be picked randomly, as long as both backends still have free capacity. The resulting allocation could then be:

| $B_1:$ | 33.3% | $B_2:$ | 33.3% |
|---|---|---|---|
| $C_2:$ | 31.0% | $C_4:$ | 4.6% |
| $C_1:$ | 2.3% | $C_1:$ | 28.7% |

First, $C_2$ is fully allocated to $B_1$, then $C_4$ is fully allocated to $B_2$ and finally the weight of $C_1$ has to be distributed onto both backends. In this example, the resulting solution will have reduced replication compared to the initial solution since $C_4$ which references all tables is only allocated to a single backend.

### 11.5.3. Local Improvement

Besides using random mutations, the knowledge about the allocation procedure can also be used to improve allocations deterministically. A well-known example of a local improvement is the 2-opt algorithm for the traveling salesman problem. It searches for crossings in routes, which obviously lead to a longer route and reorganizes the route to remove them. A similar idea can be applied on the allocation problem. One configuration that obviously leads to a non optimal allocation is the following:

| $B_1$ : 50% | $B_2$ : 50% |
|---|---|
| $C_1$ : 30% | $C_1$ : 25% |
| $C_2$ : 20% | $C_2$ : 25% |

Here two backends contain - in general among other - two common query classes. Such a configuration will not be generated by the greedy heuristic, but could be a result of random permutations. If this is the case, the query class weights can be shifted, until at least one of the backends contains only one of the two query classes. For the example above, shifting the weight of $C_2$ completely onto the second backend will lead to the improved allocation:

| $B_1$ : 50% | $B_2$ : 50% |
|---|---|
| $C_1$ : 50% | $C_1$ : 5% |
| | $C_2$ : 45% |

The general formulation is the following. If an allocation contains two backends $B_1$ and $B_2$ with more than one common query class,

$$|\{C \in \mathcal{C}|\text{assign}(C, B_1) > 0\} \cap \{C \in \mathcal{C}|\text{assign}(C, B_2) > 0\}| \geq 2 \tag{11.27}$$

then the weights of the common query classes can be shifted such that the backends have at most one query class in common. This can be done by reassigning the common query classes in a first fit manner. Since only common query classes are reassigned, the size of the fragments and the load of the backends are unimportant.

This optimization can be applied to a given allocation in $O(|\mathcal{C}|^2 \times |\mathcal{B}|)$, since for each combination of query classes all backends have to be checked. Another configuration that can be deterministically improved is the following:

| $B_1$ : 50% | $B_2$ : 50% |
|---|---|
| $C_1\{A\}$ : 45% | $C_2\{B\}$ : 45% |
| $C_3\{A, B, C\}$ : 5% | $C_3\{A, B, C\}$ : 5% |

Here two backends contain - again, in general among others - the same query class $C_3$ and another query class $C_1$ or $C_2$ that contain a subset of the relations of the replicated query class, with $C_1 \cup C_2 \subset C_3$. If the weight of $C_3$ is less than or equal to the weight of $C_1$ or $C_2$, $C_1$ or $C_2$ can be replicated instead of $C_3$, resulting in a reduced replication degree and reduced size of the allocation:

| $B_1$ : 50% | $B_2$ : 50% |
|---|---|
| $C_1\{A\}$ : 45% | $C_2\{B\}$ : 40% |
| $C_2\{B\}$ : 5% | $C_3\{A, B, C\}$ : 10% |

A generic formulation is as follows. If two backends $B_1, B_2$ contain a common query class $C_1$ and backend $B_1$ contains an other query class $C_2$ which references a subset of

the fragments in $C_1$, i.e. $C_2 \subset C_1$, and which has a higher load on backend $B_1$ than $C_1$ on backend $B_2$, then $C_2$ can be replicated instead of $C_1$. More formally, if the following constraints are satisfied:

$$C_1 \in \{C \in \mathcal{C} | \text{assign}(C, B_1) > 0 \wedge \text{assign}(C, B_2) > 0\} \tag{11.28}$$

$$C_2 \in \{C \in \mathcal{C} | \text{assign}(C, B_1) > 0 \wedge C \subset C_1\} \tag{11.29}$$

$$\text{assign}(B_1, C_2) \geq \text{assign}(B_2, C_1) \tag{11.30}$$

$C_2$ can be replicated on both backends and $C_1$ can be assigned solely to backend $B_1$. Again this optimization is not influenced by the processing power of the backends and the size of the data fragments. It lies in $O(|\mathcal{C}|^3 \times |\mathcal{B}|)$, since for every query class, two query classes have to be found which are a subset of the query class. For these three query classes all backends have to be checked. Using these two local improvement strategies, solutions can be improved deterministically.

## 11.6. Discussion

In this section we have presented an automatic allocation strategy for distributed database systems which only process requests. After a formal definition of the allocation problem we have shown that it is NP-complete. Nevertheless, for small instances an optimal solution can be computed using linear programming. Therefore, we have given an additional representation in form of a linear optimization problem.

For real problem sizes the optimal solution is not feasible, hence, we have provided a greedy algorithm. To improve the result of the greedy heuristic a meta heuristic was presented. Since some of the deficiencies of the greedy solution have a simple pattern we chose a memetic algorithm that uses deterministic improvements in combination to the randomized approach.

The allocation strategy may also be sensible for database systems with a very limited number of updates. However, for large numbers of updates it will generate a suboptimal data layout. Therefore we present an update aware definition in the next section.

# 12. Considering Updates

If the workload contains a fair number of update requests, the approach presented in chapter 12 will generate a suboptimal allocation. This is because the algorithm ignores the fact that updates have to be executed on all backends that host referenced data. Therefore, in this chapter the formal definition from above is extended to consider updates. As for the read only allocation, if a query class is allocated to a backend, all its referenced data has to be allocated completely to this backend. Furthermore, all update query classes that reference the same data as the allocated query classes also have to be allocated to the backend.

## 12.1. Formal Definition - Update Considering

As explained above in chapter 10, the basis of the allocation for workloads with updates is a classification that distinguishes between update and read requests. The formal definition of the allocation is similar to the one in section 11.1, yet some additional constraints have to be considered. In this section only new or altered definitions will be specified. Input for the allocation are two sets of query classes, update query classes $\mathcal{C}_U$ and read query classes $\mathcal{C}_Q$, where $\mathcal{C} = \mathcal{C}_Q \cup \mathcal{C}_U$. As mentioned in section 10, in general $\mathcal{C}$ is a multiset, but to simplify the equations below, we will treat it like a regular set. For an easier identification of read and update query classes, we will denote read query classes with $Q$ and update query classes with $U$ in the following.

$$\text{allocation} : \mathcal{P}(\mathcal{P}(F)) \times \mathcal{P}(F)) \rightarrow \mathcal{P}^{\mathcal{P}(F)\times[0,1]} \tag{12.1}$$

$$\text{allocation}(\mathcal{C}_Q, \mathcal{C}_U) = \mathcal{B},\ \forall C \in \mathcal{C},\ \exists B \in \mathcal{B} : f \in C \Rightarrow f \in \text{fragments}(B) \tag{12.2}$$

Again, the allocation definition does not ensure that the load is balanced. As shown in the introductory example in section 9, in the case of read and write requests a balanced load cannot be guaranteed in general for an optimal allocation. The share of the workload each backend can process is defined as above in equation 11.11. To define where query classes are allocated, the function assign, as defined in equations 11.8 and 11.9, is used. An allocation that considers updates is valid if all query classes are allocated and if every update query class is allocated to all backends that store its referenced data. The following constraints must be satisfied:

$$\forall C \in \mathcal{C}_Q : \sum_{B \in \mathcal{B}} \text{assign}(C, B) = \text{weight}(C) \tag{12.3}$$

$$\forall C \in \mathcal{C}_U, \forall B \in \mathcal{B}, C \cap \text{fragments}(B) \neq \emptyset : \text{assign}(C, B) = \text{weight}(C) \tag{12.4}$$

$$\forall C \in \mathcal{C}_U : \sum_{B \in \mathcal{B}} \text{assign}(C, B) \geq \text{weight}(C) \tag{12.5}$$

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

$$\text{updates} : \mathcal{C} \rightarrow \mathcal{P}(\mathcal{C}_U) \tag{12.6}$$

$$\text{updates}(C) = \{C_U \in \mathcal{C}_U | C \cap C_U \neq \emptyset\} \tag{12.7}$$

$$\text{updateWeight} : \mathcal{B} \times \mathcal{C} \rightarrow [0, 1] \tag{12.8}$$

$$\text{updateWeight}(B, C) = \sum_{C_U \in \text{updates}(C)} \text{assign}(C_U, B) \tag{12.9}$$

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

$$\text{assignedLoad} : \mathcal{B} \rightarrow ]0, \max_{B \in \mathcal{B}} \text{load}(B) + \sum_{C \in \mathcal{C}_U} \text{weight}(C)] \tag{12.10}$$

$$\text{assignedLoad}(B) = \sum_{C \in \mathcal{C}} \text{assign}(C, B) \tag{12.11}$$

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

$$\text{scaledLoad} : \mathcal{B} \rightarrow ]0, \max_{B \in \mathcal{B}} \text{load}(B) + \sum_{C \in \mathcal{C}_U} \text{weight}(C)] \tag{12.12}$$

$$\text{scaledLoad}(B) = \text{load}(B) * \begin{cases} scale, & \text{if } scale = max_{B' \in \mathcal{B}} \frac{\text{assignedLoad}(B')}{\text{load}(B')} > 1 \\ 1, & \text{else} \end{cases} \tag{12.13}$$

For an optimal allocation in the sense of throughput, the *scale* must be minimized. This can be ensured by the following constraint:

$$\begin{aligned} &\forall B, B' \in \mathcal{B}, B \neq B', \nexists C \in \mathcal{C}, \text{assign}(C, B) > 0 : \\ &\text{scaledLoad}(B) - \text{assignedLoad}(B) \geq \\ &\quad \text{scaledLoad}(B) - \text{assignedLoad}(B') - \\ &\quad \text{updateWeigth}(B, C_Q) + \text{updateWeigth}(B', C_Q) \end{aligned} \tag{12.14}$$

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

## 12.2. Maximum Speedup

As stated above, for a read only workload the theoretical speedup is always linear. In section 3.4, we presented methods to estimate the speedup for a fully replicated system. In this section, we will extend the formulation for partially replicated allocations. The basis is again the general formulation of Amdahl's law:

$$\text{speedup} = \frac{1}{\text{serial} + \frac{\text{parallel}}{\#\text{processors}}} \tag{12.15}$$

Serial is the serial fraction of a program, parallel the parallel fraction and #processors the number of processors the parallelized program is run on, the maximum speedup can be estimated by using the following equation:

$$\text{speedup}_{\max} \leq \frac{1}{\text{serial}} \tag{12.16}$$

To apply this to the workload of a CDBS, the serial and parallel fractions of a query workload have to be identified. Of course, the read load can be parallelized completely. The update load can also be parallelized, by allocating unrelated classes to different backends. But single update queries have to be processed completely by every backend they are allocated to. Hence, the maximum speedup of a workload is bound by:

$$\text{speedup}_{\max} \leq \frac{1}{\max_{C \in \mathcal{C}} \sum_{C_U \in \text{updates}(C)} \text{weight}(C_U)} \tag{12.17}$$

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

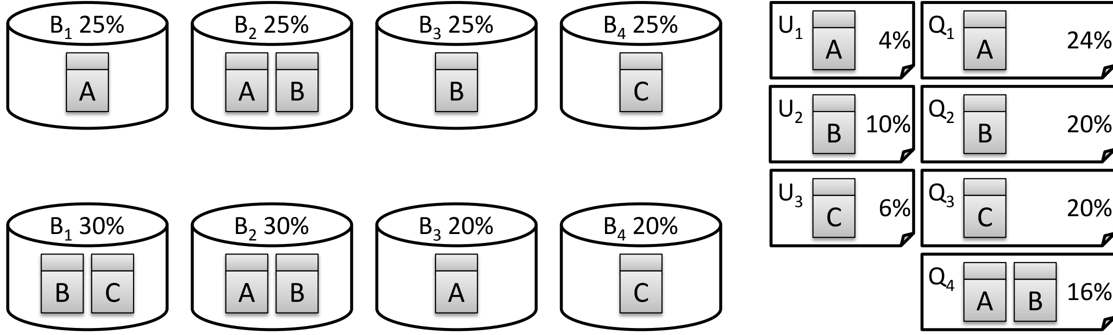



Figure 12.1.: Optimal Update Aware Allocations on Homogeneous Backends (above) and Heterogeneous Backends (below)

be interpreted as the increased workload. Using this information equation 3.14 can be modified to:

$$\text{speedup}_{hom} = \frac{1}{\text{serial} + \frac{\text{parallel}}{\#\text{processors}}} = \frac{1}{0 + \frac{scale}{\#\text{processors}}} = \frac{1}{\text{scaledLoad}} \tag{12.18}$$

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

$$\text{speedup} = \frac{1}{\frac{scale}{|\mathcal{B}|}} = \frac{|\mathcal{B}|}{scale} \tag{12.19}$$

To express the speedup in proportion to a certain backend, the computed speedup can be multiplied by the relative performance of the backend. The relative performance can be calculated by dividing the number of backends by the load of a backend:

$$\text{speedup}_{het}(B) = \frac{1}{\frac{scale}{|\mathcal{B}|}} \cdot \frac{|\mathcal{B}|}{\text{load}(B)} = \frac{1}{scale \cdot \text{load}(B)} \tag{12.20}$$

An allocation with maximum speedup can therefore be found by minimizing the maximum scaledLoad, or more generally the *scale* of an allocation.

Consider the allocations shown in figure 12.1. For the upper, homogeneous allocation the load matrix is as follows:

| | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | $U_1$ | $U_2$ | $U_3$ | Overall |
|---|---|---|---|---|---|---|---|---|
| $B_1$ | 24% | 0% | 0% | 0% | 4% | 0% | 0% | 28% |
| $B_2$ | 0% | 0% | 0% | 16% | 4% | 10% | 0% | 30% |
| $B_3$ | 0% | 20% | 0% | 0% | 0% | 10% | 0% | 30% |
| $B_4$ | 0% | 0% | 20% | 0% | 0% | 0% | 6% | 26% |

The scaledLoad of the backends is 30%, hence the *scale* is $\frac{\text{scaledLoad}(B_i)}{\text{load}(B_i)} = \frac{30\%}{25\%} = 1.2$. The speedup can be calculated using either the *scale* factor or the scaledLoad.

$$\text{speedup}_{hom} = \frac{1}{\text{scaledLoad}} = \frac{1}{30\%} = 3.\bar{3} \tag{12.21}$$

For the second, heterogeneous allocation the load matrix is:

| | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | $U_1$ | $U_2$ | $U_3$ | Overall |
|---|---|---|---|---|---|---|---|---|
| $B_1$ | 0% | 18% | 2% | 0% | 0% | 10% | 6% | 36% |
| $B_2$ | 4% | 2% | 0% | 16% | 4% | 10% | 0% | 36% |
| $B_3$ | 20% | 0% | 0% | 0% | 4% | 0% | 0% | 24% |
| $B_4$ | 0% | 0% | 18% | 0% | 0% | 0% | 6% | 24% |

The *scale* of this allocation is can be calculated with every scaledLoad:

$$\frac{\text{scaledLoad}(B_1)}{\text{load}(B_1)} = \frac{\text{scaledLoad}(B_3)}{\text{load}(B_3)} = \frac{36\%}{30\%} = 1.2 \tag{12.22}$$

The speedup can be calculated compared to the average speed of the backends:

$$\text{speedup} = \frac{|\mathcal{B}|}{scale} = \frac{4}{1.2} = 3.\bar{3} \tag{12.23}$$

It can also be calculated in comparison to one of the backends. Since the allocation has two kinds of backends, we only show this for $B_1$ and $B_3$:

$$\text{speedup}_{het}(B_1) = \frac{1}{scale \cdot \text{load}(B_1)} = \frac{1}{1.2 \cdot 0.3} = 2.\bar{7} \tag{12.24}$$

$$\text{speedup}_{het}(B_3) = \frac{1}{scale \cdot \text{load}(B_3)} = \frac{1}{1.2 \cdot 0.2} = 4.1\bar{6} \tag{12.25}$$

In the current definition, updates are considered to have an equal weight on all backends. This is correct for a *read-once/write-all* strategy for update propagation, i.e. updates are sent to all backends. In more efficient approaches, such as *primary copy*, the cost of an update will differ between the primary copy and the slaves. The formulation above can be extended to include this fact.

To ensure minimality in the number fragments equation 11.13 must be minimized, for minimal space requirements equation 11.15 must be minimized. If minimality is required, the read only allocation is a special case of this allocation. For this case, the update considering allocation is NP-hard as well. If minimality is not required, the allocation problem is still NP-hard. This is shown in the next section.

## 12.3. Proof of NP-Hardness

As stated above, if space minimality of the allocation is required, the update considering allocation is NP-hard, since the read only allocation is a special case of the update allocation. However, the allocation is still NP-hard even if the space minimality is not required. This can be shown by reducing bin packing to the allocation problem.

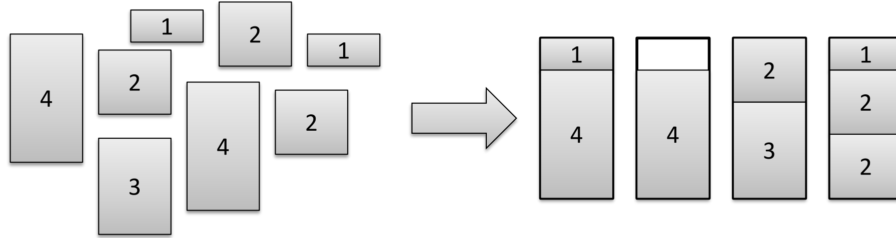



Figure 12.2.: Example of the Bin-Packing Problem

**Bin Packing** The bin packing problem decides if a set of weighted elements can be distributed amongst a set of bins with limited capacity (see figure 12.2). To give a more formal definition: given a number $n \in \mathbb{N}^+$ of bins with capacity $b \in \mathbb{N}^+$ and a set $A$ of elements $a$ with size $\text{size} : A \to \mathbb{N}^+$, where $\forall a \in A : \text{size}(a) < b$. Then the question is if $A$ can be partitioned into $n$ disjoint sets $S_1, S_2, \ldots, S_n$ such that for all $S_i$, with $1 \leq i \leq n$, the constraint $\sum_{a \in S_i} \text{size}(a) \leq b$ is satisfied? This problem is NP-complete [28]. The optimization problem to find the minimum needed $b$ is NP-hard.

*Proof.* To reduce bin packing to the allocation problem, the following configuration can be considered: let there be $n$ homogeneous backends with equal load, with $\forall B \in \mathcal{B}$ : $\text{load}(B) \leq b$. Let $\mathcal{C} = \mathcal{C}_U$ be a set of disjoint update query classes. Each query class has a weight $\text{weight}(C) < b$. Let $\text{allocation}(\mathcal{C}, m) = \mathcal{B}$ be an optimal allocation, where the maximum load is $scale * \text{load}(B)$.

In an optimal allocation $\mathcal{B}$ all query classes are allocated at least once. Furthermore, the maximum load of the backends is minimized. Since all query classes are update classes, they have to be allocated completely on the backends. Since all query classes reference disjoint data, they are independent and can be allocated independently. An optimal allocation will therefore not contain replicated query classes. Furthermore, an optimal allocation will have a minimal factor *scale*. If *scale* is less or equal to $b$, the optimal allocation is a solution to the bin packing problem. □

## 12.4. Optimal Allocation

To compute an optimal allocation again a matrix representation is used. The allocation matrix $\mathfrak{A} \in \{0, 1\}^{|\mathcal{B}| \times |F|}$ shows which data fragment is allocated to which backend. For the example in figure 9.2 the allocation matrix is:

$$\mathfrak{A} = \begin{array}{c} \\ B_1 \\ B_2 \\ B_3 \\ B_4 \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \tag{12.26}$$

The general definition is the same as to the read only case in equation 11.17. To express the load distribution of the query classes two distribution matrices are used, $\mathfrak{L}_Q \in [0,1]^{|\mathcal{B}| \times |\mathcal{C}_Q|}$ and $\mathfrak{L}_U \in [0,1]^{|\mathcal{B}| \times |\mathcal{C}_U|}$. For the example in figure 9.2, these are:

$$\mathfrak{L}_Q = \begin{array}{c} \\ B_1 \\ B_2 \\ B_3 \\ B_4 \end{array} \begin{array}{c} \begin{array}{cccc} Q_1 & Q_2 & Q_3 & Q_4 \end{array} \\ \begin{pmatrix} 0.24 & 0 & 0 & 0 \\ 0.05 & 0 & 0 & 0.16 \\ 0 & 0.20 & 0 & 0 \\ 0 & 0 & 0.20 & 0 \end{pmatrix} \end{array} \tag{12.27}$$

$$\mathfrak{L}_U = \begin{array}{c} \\ B_1 \\ B_2 \\ B_3 \\ B_4 \end{array} \begin{array}{c} \begin{array}{ccc} U_1 & U_2 & U_3 \end{array} \\ \begin{pmatrix} 0.04 & 0 & 0 \\ 0.04 & 0.10 & 0 \\ 0 & 0.10 & 0 \\ 0 & 0 & 0.06 \end{pmatrix} \end{array} \tag{12.28}$$

The general definitions of these matrices are:

$$\mathfrak{L}_Q : \begin{cases} \{1,..,|\mathcal{B}|\} \times \{1,..,|\mathcal{C}_Q|\} \rightarrow [0,1] \\ \qquad (i,k) \mapsto l_{ik} \end{cases} \tag{12.29}$$

$$\mathfrak{L}_U : \begin{cases} \{1,..,|\mathcal{B}|\} \times \{1,..,|\mathcal{C}_U|\} \rightarrow [0,1] \\ \qquad (i,k) \mapsto l'_{ik} \end{cases} \tag{12.30}$$

To ensure that all query classes are allocated the constraints 12.3 and 12.5 are satisfied. They can be formulated straightforward, as follows:

$$\forall C_k \in \mathcal{C}_Q, l_{ik} \in \mathfrak{L}_Q : \sum_{i=1}^{|\mathcal{B}|} l_{ik} = \text{weight}(C_k) \tag{12.31}$$

$$\forall C_k \in \mathcal{C}_U, l'_{ik} \in \mathfrak{L}_U : \sum_{i=1}^{|\mathcal{B}|} l'_{ik} \geq \text{weight}(C_k) \tag{12.32}$$

Again helper variables are needed to ensure that each update class is assigned to all backends, where a query class with the same data references is allocated. Analogous to equation 11.23, two matrices of helper variables are defined $\mathfrak{H}_Q$ for read query classes and $\mathfrak{H}_U$ for update classes:

$$\mathfrak{H}_Q : \begin{cases} \{1,..,|\mathcal{B}|\} \times \{1,..,|\mathcal{C}_Q|\} \rightarrow \{0,1\} \\ \qquad\qquad (i,k) \mapsto h_{ik} \end{cases}, \; h_{ik} = \begin{cases} 1, \text{ if } l_{ik} > 0 \\ 0, \text{ else} \end{cases} \tag{12.33}$$

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

$$\mathfrak{H}_U : \begin{cases} \{1,..,|\mathcal{B}|\} \times \{1,..,|\mathcal{C}_U|\} \rightarrow \{0,1\} \\ \qquad\qquad (i,k) \mapsto h'_{ik} \end{cases}$$
$$h'_{ik} = \begin{cases} 1, \text{ if } l'_{ik} > 0 \\ 1, \text{ if } \exists l_{im} \in \mathfrak{L}_Q : l_{im} > 0 \text{ and } C_k \in \text{updates}(C_m) \\ 0, \text{ else} \end{cases} \tag{12.34}$$

Using the helper variables, constraint 12.4 can be formulated as follows:

$$\forall C_k \in \mathcal{C}_U, l'_{ik} \in \mathfrak{L}_U : l'_{ik} = h'_{ik} * \text{weight}(C_k) \tag{12.35}$$

Again the load constraints of the backends must be considered. According to equation 12.13, this can be expressed using the factor *scale*:

$$\forall B_i \in \mathcal{B}, l_{ik} \in \mathfrak{L}_Q, l'_{ik} \in \mathfrak{L}_U : \sum_{k=1}^{|\mathcal{C}_Q|} l_{ik} + \sum_{m=1}^{|\mathcal{C}_U|} l'_{im} \leq scale * \text{load}(B_i) \tag{12.36}$$

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

$$\forall C_k \in \mathcal{C}_Q, \forall B_i \in \mathcal{B} : \sum_{j: f_j \in C_k} a_{ij} \geq |C_k| * h_{ik} \tag{12.37}$$

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

$$\forall C_k \in \mathcal{C}_U, \forall B_i \in \mathcal{B} : \sum_{j: f_j \in C_k} a_{ij} \geq |C_k| * h'_{ik} \tag{12.38}$$

An optimal allocation, in the sense of throughput, can be found by minimizing the factor *scale*. The definitions above can easily be translated into a linear program. In listing 12.1 a simplified instance of the example above in lp_solve syntax can be seen. The objective of the minimization is the *scale*, since all backends can process equal load, *scale* is the maximum load of a backend. In general, this would have to be multiplied by the relative load of each backend. Then the weight restrictions of the query classes and the update classes are introduced. This guarantees that each query class is fully allocated and that each update class is allocated at least once. Using the helper variables, the connection between query and update classes is established. Finally, it is guaranteed that all update classes are allocated completely to the relevant backends. However, the

Listing 12.1: Linear Program for an Update Considering Allocation

```
/* Minimization problem */
min: scale;

/* scale is maximum        */
/* weight of the backends */
lq_11 + lq_12 + lq_13 +
lq_14 + lu_11 + lu_12 +
lu_13 <= scale;
lq_21 + lq_22 + lq_23 +
lq_24 + lu_21 + lu_22 +
lu_23 <= scale;
lq_31 + lq_32 + lq_33 +
lq_34 + lu_31 + lu_32 +
lu_33 <= scale;
lq_41 + lq_42 + lq_43 +
lq_44 + lu_41 + lu_42 +
lu_43 <= scale;

/* Allocation of the */
/* query classes     */
lq_11 + lq_21 +
lq_31 + lq_41 = 0.24;
lq_12 + lq_22 +
lq_32 + lq_42 = 0.20;
lq_13 + lq_23 +
lq_33 + lq_43 = 0.20;
lq_14 + lq_24 +
lq_34 + lq_44 = 0.16;

/* Allocation of the */
/* update classes    */
lu_11 + lu_21 +
lu_31 + lu_41 >= 0.04;
lu_12 + lu_22 +
lu_32 + lu_42 >= 0.10;
lu_13 + lu_23 +
lu_33 + lu_43 >= 0.06;

/* If a query class is */
/* allocated, the      */
/* according update    */
/* classes must be     */
/* allocated           */
hu_11 - lq_11 >= 0;
hu_21 - lq_21 >= 0;
hu_31 - lq_31 >= 0;
hu_41 - lq_41 >= 0;

hu_12 - lq_12 >= 0;
hu_22 - lq_22 >= 0;
hu_32 - lq_32 >= 0;
hu_42 - lq_42 >= 0;

hu_13 - lq_13 >= 0;
hu_23 - lq_23 >= 0;
hu_33 - lq_33 >= 0;
hu_43 - lq_43 >= 0;

hu_11 - lq_14 >= 0;
hu_12 - lq_14 >= 0;
hu_21 - lq_24 >= 0;
hu_22 - lq_24 >= 0;
hu_31 - lq_34 >= 0;
hu_32 - lq_34 >= 0;
hu_41 - lq_44 >= 0;
hu_42 - lq_44 >= 0;

/* Update classes must be */
/* allocated completely   */
lu_11 - 0.04 hu_11 = 0;
lu_21 - 0.04 hu_21 = 0;
lu_31 - 0.04 hu_31 = 0;
lu_41 - 0.04 hu_41 = 0;

lu_12 - 0.10 hu_12 = 0;
lu_22 - 0.10 hu_22 = 0;
lu_32 - 0.10 hu_32 = 0;
lu_42 - 0.10 hu_42 = 0;

lu_13 - 0.06 hu_13 = 0;
lu_23 - 0.06 hu_23 = 0;
lu_33 - 0.06 hu_33 = 0;
lu_43 - 0.06 hu_43 = 0;

/* hu_ik are binary */
bin hu_11, hu_12, hu_13,
    hu_21, hu_22, hu_23,
    hu_31, hu_32, hu_33,
    hu_41, hu_42, hu_43;
```

Listing 12.2: Linear Program with Known Scale Factor

```
/* Minimization problem */
min: a_11 + a_12 + a_13
   + a_21 + a_22 + a_23
   + a_31 + a_32 + a_33
   + a_41 + a_42 + a_43;

/* Maximum load of the  */
/* backends is scale=0.3 */
lq_11 + lq_12 + lq_13 +
lq_14 + lu_11 + lu_12 +
Lu_13 <= 0.3;
lq_21 + lq_22 + lq_23 +
lq_24 + lu_21 + lu_22 +
lu_23 <= 0.3;
lq_31 + lq_32 + lq_33 +
lq_34 + lu_31 + lu_32 +
lu_33 <= 0.3;
lq_41 + lq_42 + lq_43 +
lq_44 + lu_41 + lu_42 +
lu_43 <= 0.3;

/* Allocation of the */
/* query classes     */
lq_11 + lq_21 +
lq_31 + lq_41 = 0.24;
lq_12 + lq_22 +
lq_32 + lq_42 = 0.20;
lq_13 + lq_23 +
lq_33 + lq_43 = 0.20;
lq_14 + lq_24 +
lq_34 + lq_44 = 0.16;

/* Allocation of the */
/* update classes    */
lu_11 + lu_21 +
lu_31 + lu_41 >= 0.04;
lu_12 + lu_22 +
lu_32 + lu_42 >= 0.10;
lu_13 + lu_23 +
lu_33 + lu_43 >= 0.06;

/* If a query class is */
/* allocated, the      */
/* according update    */
/* classes must be     */
/* allocated           */
hu_11 - lq_11 >= 0;
hu_21 - lq_21 >= 0;
hu_31 - lq_31 >= 0;
hu_41 - lq_41 >= 0;

hu_12 - lq_12 >= 0;
hu_22 - lq_22 >= 0;
hu_32 - lq_32 >= 0;
hu_42 - lq_42 >= 0;

hu_13 - lq_13 >= 0;
hu_23 - lq_23 >= 0;
hu_33 - lq_33 >= 0;
hu_43 - lq_43 >= 0;

hu_11 - lq_14 >= 0;
hu_12 - lq_14 >= 0;
hu_21 - lq_24 >= 0;
hu_22 - lq_24 >= 0;
hu_31 - lq_34 >= 0;
hu_32 - lq_34 >= 0;
hu_41 - lq_44 >= 0;
hu_42 - lq_44 >= 0;

/* Query classes must  */
/* be allocated on the */
/* according backend   */
hq_11 - lq_11 >= 0;
hq_12 - lq_12 >= 0;
hq_13 - lq_13 >= 0;
hq_14 - lq_14 >= 0;

hq_21 - lq_21 >= 0;
hq_22 - lq_22 >= 0;
hq_23 - lq_23 >= 0;
hq_24 - lq_24 >= 0;

hq_31 - lq_31 >= 0;
hq_32 - lq_32 >= 0;
hq_33 - lq_33 >= 0;
hq_34 - lq_34 >= 0;

hq_41 - lq_41 >= 0;
hq_42 - lq_42 >= 0;
hq_43 - lq_43 >= 0;
hq_44 - lq_44 >= 0;

/* Update classes     */
/* must be allocated */
/* completely        */
lu_11 - 0.04 hu_11 = 0;
lu_21 - 0.04 hu_21 = 0;
lu_31 - 0.04 hu_31 = 0;
lu_41 - 0.04 hu_41 = 0;

lu_12 - 0.10 hu_12 = 0;
lu_22 - 0.10 hu_22 = 0;
lu_32 - 0.10 hu_32 = 0;
lu_42 - 0.10 hu_42 = 0;

lu_13 - 0.06 hu_13 = 0;
lu_23 - 0.06 hu_23 = 0;
lu_33 - 0.06 hu_33 = 0;
lu_43 - 0.06 hu_43 = 0;

/* All fragments of a  */
/* query class must be */
/* allocated on the    */
/* according backends  */
a_11 - hq_11 >= 0;
a_21 - hq_21 >= 0;
a_31 - hq_31 >= 0;
a_41 - hq_41 >= 0;

a_12 - hq_12 >= 0;
a_22 - hq_22 >= 0;
a_32 - hq_32 >= 0;
a_42 - hq_42 >= 0;

a_13 - hq_13 >= 0;
a_23 - hq_23 >= 0;
a_33 - hq_33 >= 0;
a_43 - hq_43 >= 0;

a_11 + a_12 - 2 hq_14 >= 0;
a_21 + a_22 - 2 hq_24 >= 0;
a_31 + a_32 - 2 hq_34 >= 0;
a_41 + a_42 - 2 hq_44 >= 0;

a_11 - hu_11 >= 0;
a_21 - hu_21 >= 0;
a_31 - hu_31 >= 0;
a_41 - hu_41 >= 0;

a_12 - hu_12 >= 0;
a_22 - hu_22 >= 0;
a_32 - hu_32 >= 0;
a_42 - hu_42 >= 0;

a_13 - hu_13 >= 0;
a_23 - hu_23 >= 0;
a_33 - hu_33 >= 0;
a_43 - hu_43 >= 0;

/* hu_ik, a_ij  */
/* are binary   */
bin hq_11, hq_12,
    hq_13, hq_14,
    hq_21, hq_22,
    hq_23, hq_24,
    hq_31, hq_32,
    hq_33, hq_34,
    hq_41, hq_42,
    hq_43, hq_44,
    hu_11, hu_12, hu_13,
    hu_21, hu_22, hu_23,
    hu_31, hu_32, hu_33,
    hu_41, hu_42, hu_43,
    a_11, a_12, a_13, a_14,
    a_21, a_22, a_23, a_24,
    a_31, a_32, a_33, a_34,
    a_41, a_42, a_43, a_44;
```

resulting allocation will not be minimal in space requirements. For the example in figure 9.2, update class $U_1$ can additionally be allocated to backend $B_4$ and the theoretic throughput will still be optimal. In order to calculate a solution which is also optimal in space requirements, a second linear program is needed.

Using the first linear program, the optimal *scale* factor is calculated. In the example, the result is $scale = 0.3$. This factor is used in listing 12.2 to calculate an optimal allocation with minimal space requirements. The example is again simplified, in order to increase the readability. Therefore, the optimization goal is the number of allocated fragments and the size of the fragments is omitted.

## 12.5. Greedy Heuristic

The update considering allocation is NP-hard and hence not solvable for realistic problem sizes, therefore a heuristic is needed. The update considering heuristic is more complex than the heuristic shown in section 11.4. Since the final load of a backend cannot be calculated in advance, it has to be recalculated after every allocation of a query class. The complete algorithm can be seen in algorithm 12.1.

The heuristic starts by calculating the set $\mathcal{C}^*$:

$$\mathcal{C}^* = \mathcal{C}_Q \cup \{C_U \in \mathcal{C}_U | \nexists\, C_Q \in \mathcal{C}_Q : C_U \cap C_Q \neq \emptyset\} \tag{12.39}$$

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

For each query class in $\mathfrak{C}$ the weight that has to be assigned is stored, this is done in the while loop starting in line 6. The query class which will produce the most weight on a backend is allocated first. If all backends are already at their maximum capacity their relative load is scaled. The difference to all backends is calculated in the foreach loop starting in line 11. It is the size of the additionally allocated data fragments or 0 if the backend is empty or $\infty$ if the backend is full.

Then the query class is allocated to the backend with the least difference (lines 20 to 39). If the current query class is an update query class, the weight of the query class and the weight of its additional updates are added to the current load of the backend. The weight of the query class is set to 0, since further allocation of an update query class will result in less throughput. If the current load of the backend is larger than its scaled load, scaledLoad is adapted. We omitted the adaption of the scaled load of the other backends for the sake of brevity; this is done according to equation 12.13.

When the query class is a read query class, the maximum load of the backend has to be scaled if the backend is already full or overloaded with the updates in question. It is

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

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

**Algorithm 12.1:** Greedy Update Considering Allocation Algorithm

increased, such that the backend can hold a share matching its relative load. Then two cases have to be considered: if the complete remaining weight of the query class does not fit on the backend, as much weight of the query class as possible is assigned to the backend and the query class has to be assigned to further backends. If it does fit on the backend, the remaining weight of query class is added to the backend completely and the query class is processed completely.

At the end of the while loop (line 40) the sequence of query classes is sorted again as in line 2. When all query classes have been completely allocated the algorithm terminates.

Consider the heterogeneous allocation example in figure 12.1. It contains 7 query classes: 4 reads and 3 updates. The backends have different processing powers, while $B_1$ and $B_2$ can process 30% each, $B_3$ and $B_4$ can process 20% each.

First, the set of query classes that have to be allocated explicitly $\mathcal{C}^*$ is calculated. Since all fragments in update classes are also referenced in read classes only the read classes have to be explicitly allocated. Hence, $\mathcal{C}^* = \{Q_1, Q_2, Q_3, Q_4\}$. This set is sorted according to the weight it will produce on a backend. This is the weight of the query class itself and all referenced updates. For $Q_1$ this is weight$(Q_1)$ + weight$(U_1) = 28\%$. The result is the following list: $\mathfrak{C} = (Q_2, Q_4, Q_1, Q_3)$.

The algorithm uses the auxiliary lists currentLoad, scaledLoad and restWeight. These are initially as follows:

$$\text{currentLoad} = (B_1 : 0, B_2 : 0, B_3 : 0, B_4 : 0) \tag{12.40}$$

$$\text{scaledLoad} = (B_1 : 0.3, B_2 : 0.3, B_3 : 0.2, B_4 : 0.2) \tag{12.41}$$

$$\text{restWeight} = (Q_1 : 0.24, Q_2 : 0.2, Q_3 : 0.2, Q_4 : 0.16) \tag{12.42}$$

In the main loop, query class $Q_2$ which is first in $\mathfrak{C}$ is processed first. The difference to each backend is calculated, which is 0 for all backends, since all are empty. In the next step, all fragments of $Q_2$ and its updates (i.e. $U_2$) are allocated to $B_1$. Now the weight of the updates is allocated to $B_1$. The allocation matrix is now:

| | $A$ | $B$ | $C$ |
|---|---|---|---|
| $B_1$ | 0 | 1 | 0 |
| $B_2$ | 0 | 0 | 0 |
| $B_3$ | 0 | 0 | 0 |
| $B_4$ | 0 | 0 | 0 |

The load matrix is:

| | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | $U_1$ | $U_2$ | $U_3$ | Overall |
|---|---|---|---|---|---|---|---|---|
| $B_1$ | 0% | 0% | 0% | 0% | 0% | 10% | 0% | 10% |
| $B_2$ | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| $B_3$ | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| $B_4$ | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |

The currentLoad is updated currentLoad = $(B_1 : 0.1, B_2 : 0, B_3 : 0, B_4 : 0)$. Since $Q_2$ is a read query class, as much of the remaining weight as possible is allocated to $B_1$. In this case it is all weight of $Q_2$. currentLoad is updated, currentLoad = $(B_1 : 0.3, B_2 : 0, B_3 : 0, B_4 : 0)$ and the new load matrix is:

| | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | $U_1$ | $U_2$ | $U_3$ | Overall |
|---|---|---|---|---|---|---|---|---|
| $B_1$ | 0% | 20% | 0% | 0% | 0% | 10% | 0% | 30% |
| $B_2$ | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| $B_3$ | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| $B_4$ | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |

The allocation matrix does not change. Since $Q_2$ is completely allocated, it is removed from $\mathfrak{C}$. Hence, restWeight does not need to be updated. At the end of the loop, $\mathfrak{C}$ is sorted again: $\mathfrak{C} = (Q_4, Q_1, Q_3)$. In the next execution of the loop $Q_4$ is allocated. The differences to all backends are calculated: $(Q_4, B_1) : \infty, (Q_4, B_2) : 0, (Q_4, B_3) : 0, (Q_4, B_4) : 0$. Fragments referenced by $Q_4$, $U_1$ and $U_2$ are allocated to $B_2$. The resulting allocation matrix is:

| | $A$ | $B$ | $C$ |
|---|---|---|---|
| $B_1$ | 0 | 1 | 0 |
| $B_2$ | 1 | 1 | 0 |
| $B_3$ | 0 | 0 | 0 |
| $B_4$ | 0 | 0 | 0 |

All load of the updates ($U_1$ and $U_2$) is allocated on $B_2$ and since there is still enough load capacity for $Q_4$ it is also completely allocated on $B_2$. currentLoad is updated: currentLoad = $(B_1 : 0.3, B_2 : 0, B_3 : 0, B_4 : 0)$ The new load matrix is:

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

$Q_4$ is completely allocated and can be removed from $\mathfrak{C}$. After the sorting $\mathfrak{C}$ is $\mathfrak{C} = (Q_1, Q_3)$. In the next loop therefore $Q_1$ is allocated. The differences are: $(Q_1, B_1) : \infty, (Q_1, B_2) : \infty, (Q_1, B_3) : 0, (Q_1, B_4) : 0$. Therefore, all updates and as much weight as possible of $Q_1$ is allocated on $B_3$. The resulting allocation matrix is:

| | $A$ | $B$ | $C$ |
|---|---|---|---|
| $B_1$ | 0 | 1 | 0 |
| $B_2$ | 1 | 1 | 0 |
| $B_3$ | 1 | 0 | 0 |
| $B_4$ | 0 | 0 | 0 |

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

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

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

| | $A$ | $B$ | $C$ |
|---|---|---|---|
| $B_1$ | 0 | 1 | 0 |
| $B_2$ | 1 | 1 | 0 |
| $B_3$ | 1 | 0 | 0 |
| $B_4$ | 0 | 0 | 1 |

The resulting load matrix is:

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

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

$$\text{scaledLoad}(B_1) = \text{currentLoad}(B_1) + \text{load}(B_1) \cdot \text{weight}(Q_1) = 0.372 \tag{12.43}$$

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

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

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

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

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

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

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

| | $A$ | $B$ | $C$ |
|---|---|---|---|
| $B_1$ | 0 | 1 | 1 |
| $B_2$ | 1 | 1 | 0 |
| $B_3$ | 1 | 0 | 0 |
| $B_4$ | 0 | 0 | 1 |

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

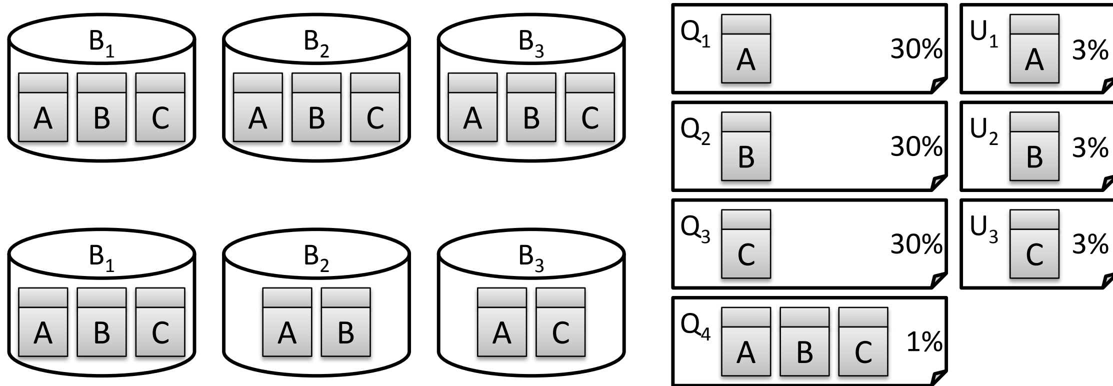



Figure 12.3.: Heuristic (above) vs Optimal (below) Allocation

The greedy heuristic does not always find the optimal allocation. An example can be seen in figure 12.3. In the example, three tables $A$, $B$ and $C$ are accessed by 7 query classes, 4 read query classes $C_{Q1}$, $C_{Q2}$, $C_{Q3}$, and $C_{Q4}$ and 3 update query classes $C_{U1}$, $C_{U2}$, and $C_{U3}$. The heuristic starts with one of $C_{Q1}$, $C_{Q2}$, and $C_{Q3}$, since all have same accumulated weight of 33%, 30% from the query class and 3% from the affected update class. Each of these classes is allocated to a single backend, since each backend can process $\frac{1}{3}$ of the workload. After this, each backend has one read and one update query class allocated. However, the remaining query class $C_{Q4}$ accesses all tables and hence cannot be allocated to a backend, without rescaling the backend load. The heuristic first allocates all update classes, which reference the same data as $C_{Q4}$ to a backend and then decides if the query class can be allocated or if the backend capacities have to be rescaled. In the rescaling phase, the accumulated load of all backends will be increased by the rest weight of the current query class. Since all backends have the same load in the example, the heuristic will allocate $\frac{1}{3}$ of $C_{Q4}$ to each backend. The result is a fully replicated allocation, with a theoretical speedup of 2.5. However, if $C_{Q4}$ is allocated to a single backend, and one of $C_{Q1}$, $C_{Q2}$, and $C_{Q3}$ is distributed as in figure 12.3, the resulting allocation can achieve a theoretic speedup of 2.7.

To improve the greedy results, a hybrid heuristic can again be applied. The general approach is same as the hybrid heuristic described in section 11.5, only the mutation and improvement strategies have to be adapted. The resulting approaches are described below.

### 12.5.1. Mutation

The mutation strategy is similar to the mutation strategy for the read only heuristic in section 11.5.2. Again, the bin packing approach is sensible. However, the strategy has to consider the changed requirements for the update query classes. The strategy randomly chooses two backends and redistributes the allocated query classes. The query classes are reallocated in a random order. Consider the non-optimal allocation in the example in

figure 12.3. Backends $B_1$ and $B_2$ have the following allocation (the percentage is given in relation to the initial load restrictions):

| $B_1$ : | 39.3% | $B_2$ : | 39.3% |
|---|---|---|---|
| $C_{Q1}$ : | 30.0% | $C_{Q2}$ : | 30.0% |
| $C_{Q4}$ : | 0.3% | $C_{Q4}$ : | 0.3% |
| $C_{U1}$ : | 3.0% | $C_{U1}$ : | 3.0% |
| $C_{U2}$ : | 3.0% | $C_{U2}$ : | 3.0% |
| $C_{U3}$ : | 3.0% | $C_{U3}$ : | 3.0% |

To redistribute the query classes, the set of all query classes that have to be allocated explicitly is generated: $\{C_{Q1}, C_{Q2}, C_{Q4}\}$. Update classes that reference the same data as read query classes in the set can be omitted, since replicating an update class will never lead to improved allocation. The set of query classes is shuffled and the query classes are allocated using a random first-fit strategy. For the sequence $(C_{Q2}, C_{Q4}, C_{Q1})$ the result is:

| $B_1$ : | 38.0% | $B_2$ : | 38.0% |
|---|---|---|---|
| $C_{Q2}$ : | 30.0% | $C_{Q4}$ : | 1.0% |
| $C_{U2}$ : | 3.0% | $C_{U1}$ : | 3.0% |
| $C_{Q1}$ : | 2.0% | $C_{U2}$ : | 3.0% |
| $C_{U1}$ : | 3.0% | $C_{U3}$ : | 3.0% |
| | | $C_{Q1}$ : | 28.0% |

Initially, $C_{Q2}$ is allocated to backend $B_1$; since $C_{U2}$ also references $B$, it is allocated as well. Then $C_{Q4}$ is allocated to backend $B_2$, with $C_{Q4}$ the update classes $C_{U1}$, $C_{U2}$, and $C_{U3}$ are allocated. Finally, $Q_{Q1}$ is allocated to both backends. As can be seen from the overall workload, the resulting allocation has a better theoretic throughput than the original solution.

### 12.5.2. Local Improvement

As for the read only case, the hybrid heuristic can employ strategies to improve the allocation deterministically. The local improvement strategies for the read only case, introduced in section 11.5.3, can also be used in the update considering allocation. Again, they have to be adapted to the update considering case.

| $B_1$ : | 60.0% | $B_2$ : | 60.0% |
|---|---|---|---|
| $C_{Q1}$ : | 30.0% | $C_{Q1}$ : | 20.0% |
| $C_{Q2}$ : | 10.0% | $C_{Q2}$ : | 20.0% |
| $C_{U1}$ : | 10.0% | $C_{U1}$ : | 10.0% |
| $C_{U2}$ : | 10.0% | $C_{U2}$ : | 10.0% |

In the example above, two query classes $C_{Q1}$ and $C_{Q1}$ are both replicated onto two backends, each references the same data as an update class. However, the referenced data of $C_{Q1}$ and $C_{Q2}$ are disjoint, as for $C_{U1}$ and $C_{U2}$. In this case, the weight of $C_{Q2}$ can be completely shifted to $B_2$ and hence $C_{U2}$, can also be deleted from $B_1$. The resulting allocation is shown below:

| $B_1$ : | 55.0% | $B_2$ : | 55.0% |
|---|---|---|---|
| $C_{Q1}$ : | 45.0% | $C_{U1}$ : | 30.0% |
| | | $C_{U2}$ : | 5.0% |
| $C_{U1}$ : | 10.0% | $C_{U1}$ : | 10.0% |
| | | $C_{U2}$ : | 10.0% |

In general, the allocation can contain further relations. This optimization can also be applied if the read query classes reference overlapping data, as long as replicated update classes can be omitted. The necessary constraints are:

$$|\{C \in \mathcal{C}_Q|\text{assign}(C, B_1) > 0\} \cap \{C \in \mathcal{C}_Q|\text{assign}(C, B_2) > 0\}| \geq 2 \tag{12.44}$$

$$\begin{gathered} C_1 \neq C_2 \in \{C \in \mathcal{C}_Q|\text{assign}(C, B_1) > 0 \wedge \text{assign}(C, B_2) > 0\} : \\ \text{updates}(C_1) \neq \text{updates}(C_2) \end{gathered} \tag{12.45}$$

If these constraints hold, it is possible to reduce the number of allocated update query classes, by shifting the query classes and therefore potentially improve the throughput. This strategy can be applied to an allocation in $O(|\mathcal{Q}|^2 \times |\mathcal{B}|)$, since for each pair of query classes all backends have to be analyzed. In some cases, where query classes are replicated, changing the distributed query class improves the allocation. Consider the following example:

| $B_1$ : | 55.0% | $B_2$ : | 55.0% |
|---|---|---|---|
| $C_{Q1}$ : | 35.0% | $C_{Q2}$ : | 33.0% |
| $C_{Q3}$ : | 5.0% | $C_{Q3}$ : | 5.0% |
| $C_{U1}$ : | 5.0% | $C_{U2}$ : | 7.0% |
| $C_{U3}$ : | 10.0% | $C_{U3}$ : | 10.0% |

In this case the query class with the least weight, $C_{Q3}$, is replicated and so is the related update class $C_{U3}$. However, since $C_{U3}$ is the heaviest update class, the resulting allocation has a non-optimal throughput. By replicating the data, which requires the least update load the allocation can be improved:

| $B_1$ : | 52.5% | $B_2$ : | 52.5% |
|---|---|---|---|
| $C_{Q1}$ : | 27.5% | $C_{Q1}$ : | 7.5% |
| $C_{Q3}$ : | 10.0% | $C_{Q2}$ : | 33.0% |
| $C_{U1}$ : | 5.0% | $C_{U1}$ : | 5.0% |
| $C_{U3}$ : | 10.0% | $C_{U2}$ : | 7.0% |

Again, the improvement concentrates on reducing the workload introduced by replicated update classes. The general formulation is:

$$C_{U1} \in \{C \in \mathcal{C}_U | \text{assign}(C, B_1) > 0 \wedge \text{assign}(C, B_2) > 0\} \tag{12.46}$$

$$C_{U2} \in \{C \in \mathcal{C}_U | \text{assign}(C, B_1) > 0\} : \text{weight}(C_{U2}) < \text{weight}(C_{U1}) \tag{12.47}$$

$$\sum_{\{C \in \mathcal{C}_Q | C \cap C_{U2} \neq \emptyset\}} \text{assign}(C, B_1) \geq \sum_{\{C \in \mathcal{C}_Q | C \cap C_{U1} \neq \emptyset\}} \text{assign}(C, B_2) \tag{12.48}$$

$$\sum_{C \in \bigcup_{\{C' \in \mathcal{C}_Q | \text{assign}(C', B_1) > 0 \wedge C' \cap C_{U2} \neq \emptyset\}} \text{updates}(C')} \text{weight}(C) < \text{weight}(C_{U1}) \tag{12.49}$$

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

## 12.6. Discussion

In this section we have presented an allocation algorithm that considers the influence of updates on the performance of a cluster database system. Due to the additional dependencies it is considerably more complex than the read only version presented in section 11. We have extended the definition of the read only version and again shown that it is NP-complete. Since replicated write query classes increase the workload the speedup of a distributed system with write access is usually not perfect. Hence, we have given equations to calculate the maximum achievable speedup for a given configuration. This definition also enables an evaluation of the performance of an implementation. It is also the basis for deciding if a certain configuration can achieve the needed throughput. Our test results show that the maximum speedup is also a good indicator for the effective speedup. This will be shown in the evaluation in section 15.2.

We extended the linear program for the calculation of an optimal allocation and presented a more sophisticated greedy heuristic. To improve the results of the greedy heuristic we adapted the memetic algorithm and the improvement strategies.

In the next section we will extend both the read only approach and the read write approach to enhance the reliability of a cluster database system. This is done by ensuring that each fragment or query class is replicated several times in the cluster.

# 13. K-Safety

In distributed systems error rates are multiplied by the number of components. Therefore, in large scale clusters failures are daily business. For example, Google published numbers that revealed each of their clusters of 1800 nodes had over 1000 machine failures in the first year and thousands of hard disk failures [203]. Hence, distributed systems have to include mechanisms to deal with hardware failures. In this chapter, we will limit the discussion to the treatment of hardware or system failures that can be narrowed down to a single backend or a group of backends. For systems that require high availability this is obviously not enough; apart from hardware failures on the backends, master hardware failures and software failures also have to be taken in consideration.

On the cluster level, the standard fault tolerance approach is redundancy [107, 191]. Therefore, we will present extensions of our algorithms that introduce k-safety [211]. With k-safety, the algorithms ensure that the loss of $k$ backends can be tolerated. If each fragment is allocated to at least $k+1$ backends, no data is lost if $k$ backends fail. However, to ensure that all queries can still be processed locally without a reallocation, each query class has to be allocated to at least $k+1$ backends. The basis is still the CDBS processing model introduced in section 3.4; hence, updates can only be processed on backends that have all referenced data. As a result, the independent allocation of fragments that are updated is not possible. According to equation 12.4 update query classes have to be allocated completely to each backend.

In the read only case, the introduction of k-safety only has the drawback of increased space requirements. Obviously, the theoretical speedup is unaffected by additional replicas. On the contrary, additional replicas allow a more flexible load balancing, especially if the load is slightly varying. In practice, however, the additional replicas will result in a less fragmented schema and hence larger relations. In some cases, especially for large relations, this will reduce the performance due to an increased cache miss rate. For the update sensitive case, the replication reduces the performance if the replicas introduce replicated updates.

The formal definitions and algorithms presented before can easily be adapted. First we will explain the extensions for the allocation of $k+1$ replicas of each fragment for the read only case and then the allocation of each query class to $k+1$ backends for both cases.

## 13.1. Redundant Fragments

As explained above, the independent replication of fragments without replicating complete query classes is in general only possible for read only access of data. However, in the presence of updates the read only fragments can still be replicated without considering the

query classes. We will not elaborate on this option, since the approach is a straightforward combination of the fragment and query class replication.

If $k+1$ copies of each fragment have to be allocated, the following constraint can be introduced to the formal definition in section 11.1:

$$\forall f \in F : \sum_{\{B \in \mathcal{B} | f \in \text{fragments}(B)\}} 1 \geq k+1 \tag{13.1}$$

In the matrix definition (and hence the linear program), the constraint can be formulated using the allocation matrix $\mathfrak{A}$ as defined in 11.16:

$$\forall i \in \{1, ..., |B|\}, a_{ij} \in \mathfrak{A} : \sum_{j=1}^{|F|} a_{ij} \geq k+1 \tag{13.2}$$

Both formulations ensure that each fragment is allocated at least $k+1$ times. Without further restrictions, a straightforward solution in the read only case is to place fragments that have to be further allocated to the first backends to which they are not already allocated. To ensure a better distribution of the additional data, a randomized approach can be used. In the update sensitive case, only fragments that are never updated can be freely placed. The placement of fragments with updates is part of the optimization goal.

The read only heuristic (algorithm 11.1) can be extended by an additional loop after the main foreach loop (starting in line 4) that ensures the k-safety. The pseudo-code can be seen in algorithm 13.1.

```
30 foreach f ∈ F do
31 |   𝔅 ← {B ∈ ℬ | f ∈ fragments(B)};
32 |   while |𝔅| < k + 1 do
33 |   |   B ← random element(ℬ \ 𝔅);
34 |   |   fragments(B) ← fragments(B) ∪ {f};
35 |   |   𝔅 ← 𝔅 ∪ {B};
36 |   end
37 end
```

**Algorithm 13.1:** K-Safety Loop for the Read Only Greedy Allocation Algorithm

## 13.2. Redundant Query Classes

To ensure that the CDBS is fully operational after the loss of $k$ backends, each query class has to be allocated to at least $k+1$ backends. In the formal definitions of the read only case (see section 11.1) and the update case (see section 12.1), the following constraint ensures k-safety:

$$\forall C \in \mathcal{C} : \sum_{B \in \mathcal{B}} \begin{cases} 1, \text{ if assign}(C, B) > 0 \\ 0, \text{ else} \end{cases} \geq k + 1 \tag{13.3}$$

For the matrix definition and hence the linear program, the constraint can be neatly defined using the helper variables in $\mathfrak{H}$ for the read only case[1]:

$$\forall i \in \{1, ..., |B|\}, h_{ij} \in \mathfrak{H} : \sum_{j=1}^{|\mathcal{C}|} h_{ij} \geq k + 1 \tag{13.4}$$

The definition of the constraints in the update case for the matrices $\mathfrak{H}_Q$ and $\mathfrak{H}_U$ is analog.

The adaption of the read only heuristic can be done similarly to the fragment replication shown above as show in algorithm 13.2.

```
30 foreach C ∈ 𝒞 do
31   𝔅 ← {B ∈ ℬ|C \ fragments(B) ≠ ∅};
32   while |𝔅| < k + 1 do
33     B ← random element(ℬ \ 𝔅);
34     fragments(B) ← fragments(B) ∪ C;
35     𝔅 ← 𝔅 ∪ {B};
36   end
37 end
```

**Algorithm 13.2:** K-Safety Loop for the Read Only Greedy Allocation Algorithm

Since replicated updates increase the workload and thus potentially reduce the overall throughput, the adaption of the update heuristic has to be more sophisticated. Therefore, the replicated query classes are treated like other query classes and are allocated accordingly. This can be done by introducing a new set $\mathcal{C}_k$, which contains the query classes that have to be further replicated. Initially, $\mathcal{C}_k$ is empty. Whenever a query class $C \in \mathcal{C}_Q$ is completely allocated, the number of backends that it has been allocated to is counted and stored in *replicas* and if $replicas < k+1$, then $k+1-replicas$ copies of $C$ are added to $\mathfrak{C}$ and $\mathcal{C}_k = \mathcal{C}_k \cup \{C\}$. This can be done by the pseudo-code shown in algorithm 13.3.

These replicated queries all have no weight, except for the update classes that have to be allocated additionally. Hence, they can be treated like update classes and be allocated each to a single backend. If the additional updates overload the backend the scaledLoad has to be adapted, hence the condition of the if statement in line 23 is altered to $C \in \mathcal{C}_U \vee C \in \mathcal{C}_k$. Finally, it has to be assured that replicated query classes are not allocated to backends which already contain a replica. This can be done by setting the according difference to $\infty$.

[1] In order to avoid the name conflict of the index $k$ for query classes and k-safety, we changed the query class index to $j$.

35 $\mathfrak{B} \leftarrow \{B \in \mathcal{B} | C \setminus \text{fragments}(B) \neq \emptyset\}$;
36 $replicas \leftarrow |\mathfrak{B}|$;
37 **if** $replicas < k + 1$ **then**
38 $\quad \mathfrak{C} \leftarrow \mathfrak{C} \cup ((k + 1 - replicas) \cdot \mathcal{C}_k)$;
39 $\quad \mathcal{C}_k = \mathcal{C}_k \cup \{C\}$;
40 **end**

**Algorithm 13.3:** Pseudo Code for Adding Missing Replicas of Query Classes

To ensure that all update classes are allocated $k+1$ times, those that are not allocated with query classes have to be added $k + 1$ times to $\mathfrak{C}$. The complete pseudo code can be seen in algorithm 13.4.

## 13.3. Discussion

In this section we have extended our allocation approach to ensure k-safety. Replicating every data fragment multiple times reduces the probability of data loss enormously. This is equivalent to the increased reliability of a RAID system [174]. Since our processing model enforces local execution of query classes k-safety of data fragments is not enough to ensure that a system is still capable of processing all queries. Hence, we have presented a second definition of k-safety which ensures that a CDBS can tolerate the loss of k backends and still process all incoming queries. For write access this is the only valid definition of k-safety, since our model does not allow distributed update processing.

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

1. $\mathcal{C}^* \leftarrow \mathcal{C}_Q \cup \{C_U \in \mathcal{C}_U | \nexists\, C_Q \in \mathcal{C}_Q : C_U \cap C_Q \neq \emptyset\}$;
2. $\mathcal{C}_k \leftarrow \{C_U \in \mathcal{C}_U | \nexists\, C_Q \in \mathcal{C}_Q : C_U \cap C_Q \neq \emptyset\}$;
3. $\mathfrak{C} \leftarrow$ sort $\mathcal{C}^* \cup (k \cdot \mathcal{C}_k)$ descending to $\text{weight}(C) + \sum_{C_U \in \mathcal{C}_U} \begin{cases} 0, & C = C_U \\ \text{weight}(C_U), & \text{if } C \cap C_U \neq \emptyset \\ 0, & \text{else} \end{cases}$;
4. $\text{currentLoad}(\mathcal{B}) \leftarrow 0$;
5. $\text{scaledLoad}(\mathcal{B}) \leftarrow \text{load}(\mathcal{B})$;
6. $\text{restWeight}(\mathfrak{C}) \leftarrow \text{weight}(\mathfrak{C})$;
7. **while** $C \in \mathfrak{C}$ **do**
8.     **if** *all backends are full* **then**
9.         **foreach** $B \in \mathcal{B}$ **do**
10.             $\text{scaledLoad}(B) \leftarrow \text{currentLoad}(B) + \text{load}(B) \cdot \text{weight}(C)$;
11.         **end**
12.     **foreach** $B \in \mathcal{B}$ **do**
13.         **if** $(\mathit{currentLoad}(B) = \mathit{scaledLoad}(B)) \vee (\mathit{assign}(B, C) > 0)$ **then**
14.             $\text{difference}(C, B) \leftarrow \infty$;
15.         **else if** $\mathit{currentLoad}(B) = 0$ **then**
16.             $\text{difference}(C, B) \leftarrow 0$;
17.         **else**
18.             $\text{difference}(C, B) \leftarrow \text{size}((C \cup \text{updates}(C)) \setminus \text{fragments}(B))$;
19.         **end**
20.     **end**
21.     $B \leftarrow B \in \mathcal{B}$ with $\text{difference}(C, B)$ minimal;
22.     $\text{fragments}(B) \leftarrow \text{fragments}(B) \cup C \cup \text{updates}(C)$;
23.     $\text{currentLoad}(B) \leftarrow \text{currentLoad}(B) + \text{weight}(\text{updates}(C)) - \text{updateWeight}(B, C)$;
24.     **if** $(C \in \mathcal{C}_U) \vee (C \in \mathcal{C}_k)$ **then**
25.         **if** $\mathit{currentLoad}(B) > \mathit{scaledLoad}(B)$ **then**
26.             $\text{scaledLoad}(B) \leftarrow \text{currentLoad}(B)$;
27.         **end**
28.         $\mathfrak{C} \leftarrow \mathfrak{C} \setminus \{C\}$;
29.     **else**
30.         **if** $\mathit{currentLoad}(B) \geq \mathit{scaledLoad}(B)$ **then**
31.             $\text{scaledLoad}(B) \leftarrow \text{currentLoad}(B) + \text{load}(B) \cdot \text{weight}(C)$;
32.         **end**
33.         **if** $\mathit{restWeight}(C) > \mathit{scaledLoad}(B) - \mathit{currentLoad}(B)$ **then**
34.             $\text{restWeight}(C) \leftarrow \text{restWeight}(C) - (\text{scaledLoad}(B) - \text{currentLoad}(B))$;
35.             $\text{currentLoad}(B) \leftarrow \text{scaledLoad}(B)$;
36.         **else**
37.             $\text{currentLoad}(B) \leftarrow \text{currentLoad}(B) + \text{restWeight}(C)$;
38.             $\mathfrak{C} \leftarrow \mathfrak{C} \setminus \{C\}$;
39.             $\mathfrak{B} \leftarrow \{B \in \mathcal{B} | C \setminus \text{fragments}(B) \neq \emptyset\}$;
40.             $\mathit{replicas} \leftarrow |\mathfrak{B}|$;
41.             **if** $\mathit{replicas} < k+1$ **then**
42.                 $\mathfrak{C} \leftarrow \mathfrak{C} \cup ((k + 1 - \mathit{replicas}) \cdot \mathcal{C}_k)$;
43.                 $\mathcal{C}_k = \mathcal{C}_k \cup \{C\}$;
44.             **end**
45.         **end**
46.     **end**
47.     $\text{sort}(\mathfrak{C})$ descending to restWeight;
48. **end**
49. **return** $\mathcal{B}$

**Algorithm 13.4:** Greedy Update Considering Allocation Algorithm with K-Safety

# 14. Physical Allocation

The allocation algorithm presented before calculates an allocation based only on the query history and the cluster environment. In most cases, the allocation will be calculated for a system which already has several backends. The cluster allocation algorithm does not take a previous allocation into account. Therefore, the new allocation has to be implemented in the running database cost efficiently. Several factors contribute to the cost of a physical allocation. Basically, it is again an ETL process (cf. section 4.1), so data extraction, data transport and data loading have to be considered. Obviously, data that is already allocated to a certain backend will not create any costs. However, data that has to be transferred to a new backend and imported into the database system will create noticeable costs. This cost is mainly related to the size of the data. Hence, a good approach is to reduce the amount of transfered data. In order to implement the allocation in the system in a cost optimal way according to this measure, a matching between the newly calculated allocation and currently installed allocation has to be found. The problem can be modeled using a complete, weighted bipartite graph $G = (B' \cup B, E)$. Nodes $B'$ represent the backends in the new allocation and nodes $B$ the backends in the old allocation. Both node sets have the same size $n$. Each node in $B'$ is connected with every node in $B$ with an edge $e \in E$. The weight of an edge $e_{vu}$ between node $B'_v \in B'$ and node $B_u \in B$ is the cost of allocating the data fragments in $B'_v$ to $B_u$. Usually, this cost is mainly dependent of the size of the data that has to be transferred and imported. Hence, in many cases a valid approximation of the weight is the sum of the sizes of data fragments which have to be additionally allocated to the backend. This can be calculated by the following equation:

$$\text{weight}(e_{uv}) = \sum_{f \in \text{fragments}(B'_v) \setminus \text{fragments}(B_u)} \text{size}(f) \tag{14.1}$$

This may not be true for all applications. If the import of data fragments has varying costs, for example from the use of different storage structures and indices, the size of the fragments alone will not be an adequate measurement. In heterogeneous clusters the transport costs of the data may also vary, so the network topology can further complicate the calculation.

Consider the example in figure 14.1. A cluster database with three backends is shown, where $B_1$ stores table $A$, $B_2$ tables $A$ and $B$ and $B_3$ table $C$. A new allocation was calculated, where one $B'_1$ has to store table $A$, $B'_2$ stores tables $B$ and $C$ and $B'_3$ stores tables $A$ and $B$. The costs for implementing allocation $B'_v$ to backend $B_u$ is the size of the data that has to be transferred to backend $B_u$. For example, allocating $B'_2$ to backend $B_2$ will introduce costs relative to the size of table $C$, since $C$ is not yet allocated to

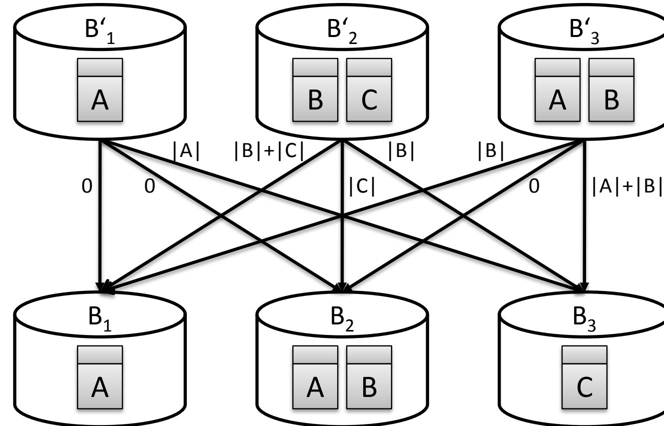



Figure 14.1.: Complete Bipartite Graph of the New Allocation (above) and the Existing Configuration (below)

backend $B_2$. A cost optimal implementation of the new allocation can be found using an optimal matching between the new and old allocation. In the example, a cost optimal implementation of the new allocation can be found by matching $B'_1$ with $B_1$, $B'_2$ and $B_3$, and $B'_3$ with $B_2$ as shown in figure 14.2. It introduces costs relative to the size of table $B$, since only this table has to be transferred.

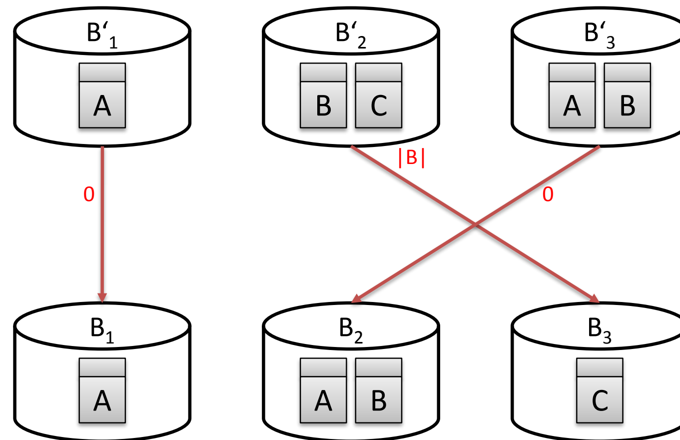



Figure 14.2.: Optimal Matching of the New Allocation (above) and the Existing Configuration (below)

This matching is a common problem and it is known as the assignment problem [51]. It is a special form of linear optimization which is strongly polynomial. To generate a cost-minimal perfect matching the Hungarian method, also known as Kuhn-Munkres algorithm, is used [143, 159]. This algorithm calculates an optimal matching in $\mathcal{O}(n^3)$. We will explain the function of the Hungarian method by the example in figure 14.3.

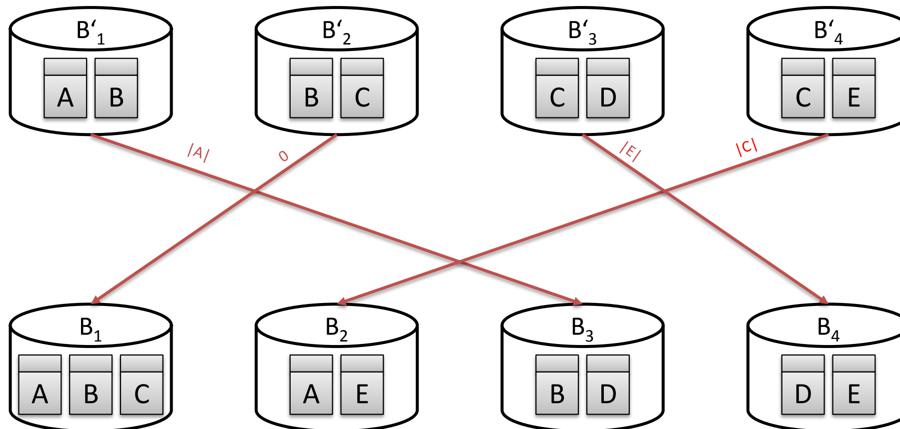



Figure 14.3.: Optimal Matching for a New Allocation (above) and an Existing Configuration (below)

An allocation for 4 backends is calculated, where the sets of fragments in the new allocation are: $\{AB\}$, $\{BC\}$, $\{CD\}$, and $\{CE\}$. The sets of already allocated fragments are: $\{ABC\}$, $\{AE\}$, $\{BD\}$, and $\{DE\}$. Let the sizes of the tables be: size$(A) = 1$, size$(B) = 2$, size$(C) = 3$, size$(D) = 4$, and size$(E) = 5$. The Hungarian method uses a matrix representation of the complete bipartite graph of the sets of backends. The entries are the costs of matching a backend of the new allocation to an existing backend. The cost is calculated with equation 14.1:

$$\begin{array}{c} \\ B'_1 \\ B'_2 \\ B'_3 \\ B'_4 \end{array} \begin{array}{c} \begin{array}{cccc} B_1 & B_2 & B_3 & B_4 \end{array} \\ \begin{pmatrix} 0 & |B| & |A| & |A|+|B| \\ 0 & |B|+|C| & |C| & |B|+|C| \\ |D| & |C|+|D| & |C|+|D| & |C| \\ |E| & |C| & |C|+|E| & |C| \end{pmatrix} \end{array} = \begin{array}{c} \\ B'_1 \\ B'_2 \\ B'_3 \\ B'_4 \end{array} \begin{array}{c} \begin{array}{cccc} B_1 & B_2 & B_3 & B_4 \end{array} \\ \begin{pmatrix} 0 & 2 & 1 & 3 \\ 0 & 5 & 3 & 5 \\ 4 & 7 & 7 & 3 \\ 5 & 3 & 8 & 3 \end{pmatrix} \end{array} \tag{14.2}$$

In this matrix a matching can be found by selecting entries such that for each row one entry is selected and that no other row's element is in the same column. If the sum of the selected elements is minimal the matching is optimal. Obviously, there are $n!$ possible matchings. The Hungarian method begins with searching the minimum of each row and from the other elements in the same row. After this step each row has at least one element that is 0:

$$\begin{array}{c} \\ B'_1 \\ B'_2 \\ B'_3 \\ B'_4 \end{array} \begin{array}{c} \begin{array}{cccc} B_1 & B_2 & B_3 & B_4 \end{array} \\ \begin{pmatrix} 0 & 2 & 1 & 3 \\ 0 & 5 & 3 & 5 \\ 1 & 4 & 4 & 0 \\ 2 & 0 & 5 & 0 \end{pmatrix} \end{array} \tag{14.3}$$

If there is at least one 0 in each column then an optimal matching can be selected. In the example above this is not possible since for column $B_3$ no element is 0. Therefore, for each column the minimum is searched and subtracted from the other elements in that column:

$$\begin{array}{c} \\ B'_1 \\ B'_2 \\ B'_3 \\ B'_4 \end{array} \begin{array}{c} \begin{array}{cccc} B_1 & B_2 & B_3 & B_4 \end{array} \\ \begin{pmatrix} 0 & 2 & \underline{0} & 3 \\ \underline{0} & 5 & 2 & 5 \\ 1 & 4 & 3 & \underline{0} \\ 2 & \underline{0} & 4 & 0 \end{pmatrix} \end{array} \tag{14.4}$$

In the example an optimal matching can be found by matching $B'_1$ to $B_3$, $B'_2$ to $B_1$, $B'_3$ to $B_4$, and $B'_4$ to $B_2$ (indicated by the underlined 0's). In general the second step might not directly lead to a result. If this is the case, then the minimum number of rows and columns is selected that covers all 0's. The minimum value of the unselected elements is searched and subtracted from all unselected elements and added to all selected elements. The resulting matrix is used as input for the Hungarian method again. This procedure is repeated until a matching can be found.

The matching only makes sense for homogeneous clusters, since each backend of the allocation can be mapped to any physical backend; for heterogeneous clusters the algorithm calculates an allocation that is adapted to the different capacities of the backends. If every node in the system has different processing capabilities, then the calculated allocation is tailored to the capabilities of each backend and hence no matching is necessary. As clusters usually comprise several groups of nodes with equal hardware, optimal matchings within these clusters can be calculated.

## 14.1. Implementing Scaling

Using the matching, a cost optimal scaling of the database system can also be calculated. If a system is scaled, a new allocation is calculated and either new nodes are added or removed. The data mapping again can be modeled using a complete bipartite graph $G = (B' \cup B, E)$. In both situations the cardinality of $B'$ and $B$ is different: in the case of an increase in the number of nodes, $|B'| > |B|$ and in the case of a scale down $|B'| < |B|$. In order to use the Hungarian method, both sets have to have the same number of nodes. In the case of a scale out, the old allocation is simply increased with empty virtual backends. These present the new, unpopulated nodes. If the system is scaled down, it might be reasonable to remove a certain set of backends which have less disk capacity or bad connectivity or the like. In this case, the backends concerned are removed and the matching is done as if there was no scaling. If the backends to be removed can be freely chosen from all backends, the new allocation is extended with empty backends and the matching is done. The backends of the old allocation that are matched with the empty backends will then be removed from the cluster.

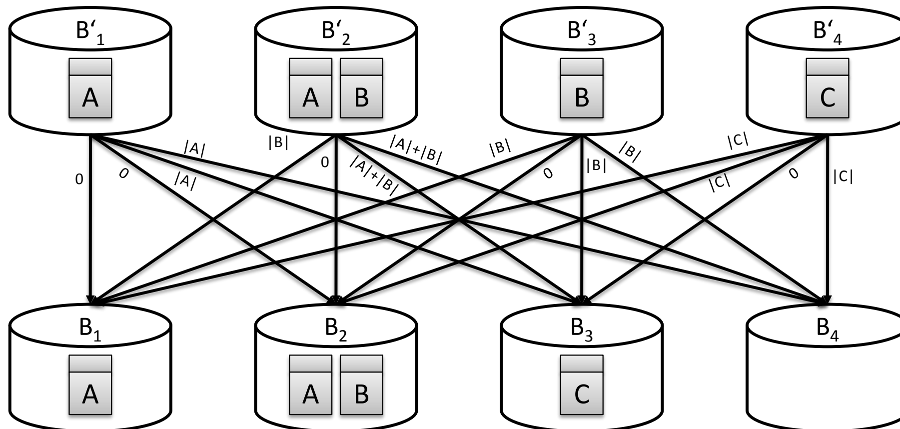



Figure 14.4.: Complete Bipartite Graph for the Mapping between a Scaled Allocation (above) and a New Hardware Configuration (below)

In figure 14.4, an example of a scale-out can be seen. The cluster database system is scaled from three to four nodes. Obviously, the new backend $B_4$ is initially empty. The new allocation already has four backends, so the matching is done in the same way as above, using the Hungarian method. The result of the Hungarian method can be seen in figure 14.5.

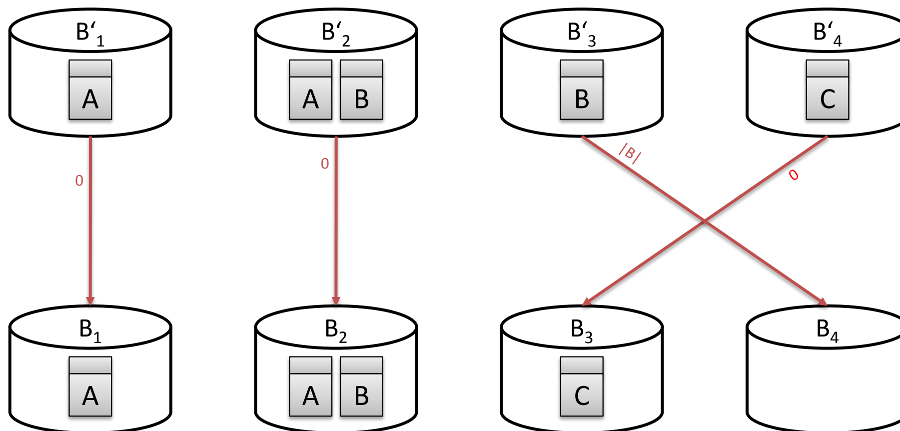



Figure 14.5.: Optimal Mapping between a Scaled Allocation (above) and a New Hardware Configuration (below)

## 14.2. Discussion

Using the Hungarian method a cost optimal implementation of a calculated allocation can be found. In this section we limited the cost metric to the size of the transferred data.

Using more sophisticated cost metrics would make it possible to take additional factors into account. For example, if the cost of the data import is non-linear to the size of the transferred data, the presented metric is not exact. For vertical partitioning the adding of columns as well as the subtraction of columns results in additional costs that are related to the size of the sum of the fragments, which also could be taken into account.

As mentioned before, for completely heterogeneous systems the calculated allocation is already tailored to each backend. This makes the matching unnecessary. However, many heterogeneous systems consist of multiple subgroups of similar backends. Within these groups the Hungarian method can be used to minimize the cost of the physical allocation.

# 15. Evaluation

We implemented two prototypes to test the allocation algorithms. In the following we will discuss the architecture of the two prototypes and then present the results.

The first prototype was based on the Sequoia middleware (see section 3.5.1). We added methods to adjust the allocation of the relations and degree of replication and methods to scale a cluster at runtime. Our software communicates via the available JMX/RMI interface with the Sequoia controller. This approach made it possible to manage a cluster consisting of more than one controller and to keep changes to the Sequoia code base as small as possible. An overview of the architecture can be seen in figure 15.1. Our scalable cluster management tool (SCMT) consists of three main components: a monitoring component, a strategy component, which implements the allocation algorithm presented in chapter 11, and an execution component, which executes all actions on the controller or the database backends. Monitoring includes collecting data from the backend database systems (i.e. memory- and CPU-utilization and available disk space) and gathering statistics of requests submitted to the cluster. This information is stored in an embedded database at the controller for further analysis. The strategy component has two main functions. First, it decides whether the cluster should be resized by adding a new database backend or by dropping an existing one as explained in section 6.4. The second function is to calculate the allocation. The execution components perform the set of actions identified by the strategy component. It contains a simple task scheduler which executes a series of actions in parallel. This may include adding a new node to cluster, removing a node, setting up the database system on the node and replicating or deleting data on the database backends. To improve performance, database specific tools such as bulk loaders etc. are used. To enable an automatic installation of the backend system, the database management system Apache Derby was used. Apache Derby is a lightweight database management system that is completely implemented in Java. It has a footprint of 2 megabytes and needs no installation.

The prototype is a fully functional cluster database management system. However, it has several limitations due to the use of third party software components. The Sequoia system was primarily designed for full replication. It supports the RAIDb-2 level, which allows a partial replication of the database. However, no partitioning is supported. This limits the possibilities of the allocation algorithm. Furthermore, the backend DBMS Apache Derby has problems with large data sizes. In our tests, it was not possible to run a test with a one gigabyte database. Therefore, we implemented a second prototype.

The second prototype was implemented to only test the allocation algorithms. It consists of a controller that encapsulates all logic. The architecture can be seen in figure 15.2. The controller has two modes of operation: allocation and query processing. In the query

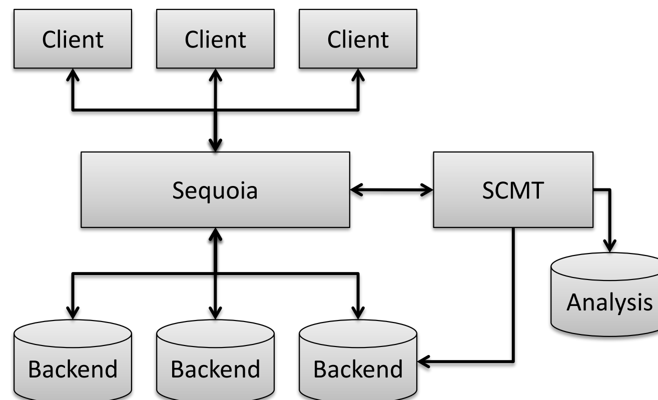



Figure 15.1.: Architecture of the First Prototype

processing mode, the controller starts the driver that issues SQL requests. The requests are sent to the scheduler that holds a queue for each backend. The scheduler inserts incoming requests into the queues according to the *least pending request first* order. If the database is partially replicated across the backends, the scheduler also decides which backend can handle a request. The data allocation is stored in the schema. For each queue multiple connections are opened to the according database system and each connection holds a single request at a time. PostgreSQL[1] and MySQL[2] are used as backend database systems. In order to utilize the backend system at least one connection per processor core of the backend system should be started, since both PostgreSQL and MySQL start one thread per connection. Each processed request is stored in the query history along with its processing time. Furthermore, statistical data is stored in an embedded database for later analysis.

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

In order to guarantee comparable results, we do not further optimize the data layout. Therefore, the schema only includes indexes that are generated automatically (i.e. indexes on the primary keys). All tests are run on a 16 node high performance computing cluster. Each node has two Intel Xeon QuadCore processors with 2 GHz clock rate, 16 GB RAM and two 74 GB SATA hard disks with a RAID 0 configuration. We use separate nodes for the controller and the database backends. We tested our algorithms with TPC-H and TPC-App style benchmarks.

[1]PostgreSQL - `http://www.postgresql.org` (last visited 2011-04-15)
[2]MySQL - `http://www.mysql.com/` (last visited 2011-04-15)

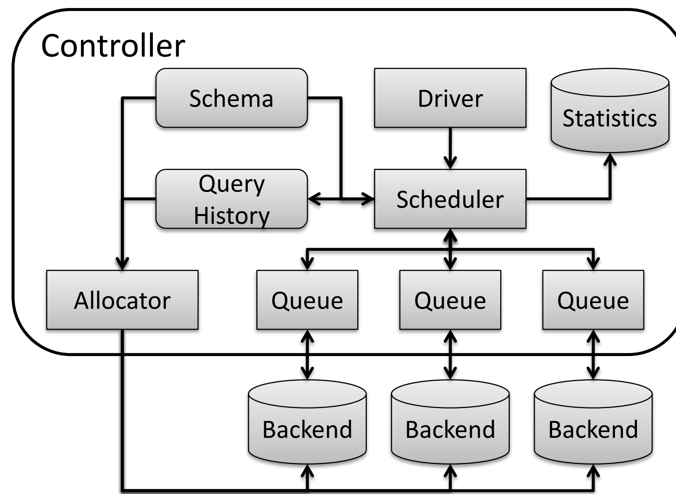



Figure 15.2.: Architecture of the Second Prototype

## 15.1. TPC-H

TPC-H is a decision support benchmark; a detailed description is given in section 17.1. Even though TPC-H defines update statements, we used the benchmark without these to test our read only algorithm. PostgreSQL and MySQL have problems with the complexity of the queries in TPC-H. Since queries 17 and 20 are disproportionally slow in PostgreSQL, we omitted them in the test. We used scale factor 1, which results in a 1 GB data set, since larger scale factors further reduce the number of processable queries. We ran tests for full replication, table based allocation and column based allocation. Each test consisted of 10 runs on each number of backends. For the allocation test, we started with full replication for each scale in order to get an initial weight distribution for the queries. In each test 10000 queries were sent to the database, which corresponds to 500 variations of each query. In figure 15.3 the average of the runs can be seen.

It can be seen that all configurations scale linearly. Furthermore, both, the table and the column based allocation, outperform the full replication. For table based allocation, this is due to the improved caching on the backend database systems. Since the query classes differ strongly in their weight, some heavy classes are allocated exclusively to multiple backends. Because the backends are specialized on single query classes, less data is stored on the nodes and hence the caching on these backends improves. For column based allocation the throughput further increases, since the vertical partitioning improves the data transfer speed from the disk. Obviously, the quality of the allocation is highly dependent on the quality of the classification and estimation of the weight of the query classes. In figure 15.4 the minimum and maximum throughput of the column based allocation in the 10 test runs is shown. Although the column based allocation has the largest deviation in throughput, it is still never above 6%. This shows that the sum of the execution times of the queries is an excellent measure for the weight of a query class.

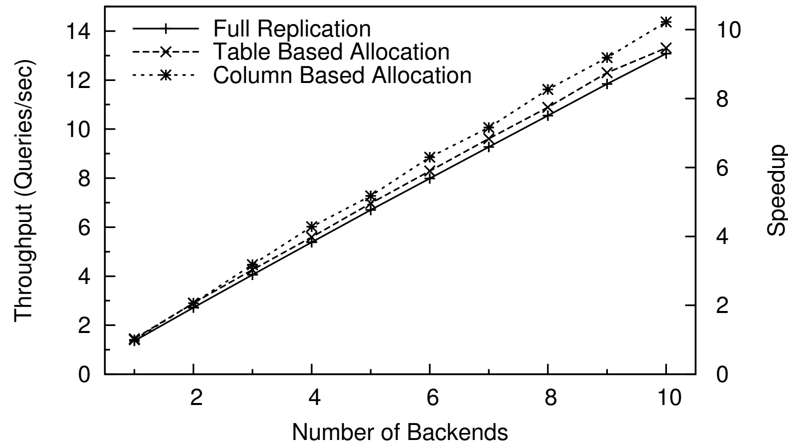



Figure 15.3.: TPC-H Throughput for Different Cluster Sizes

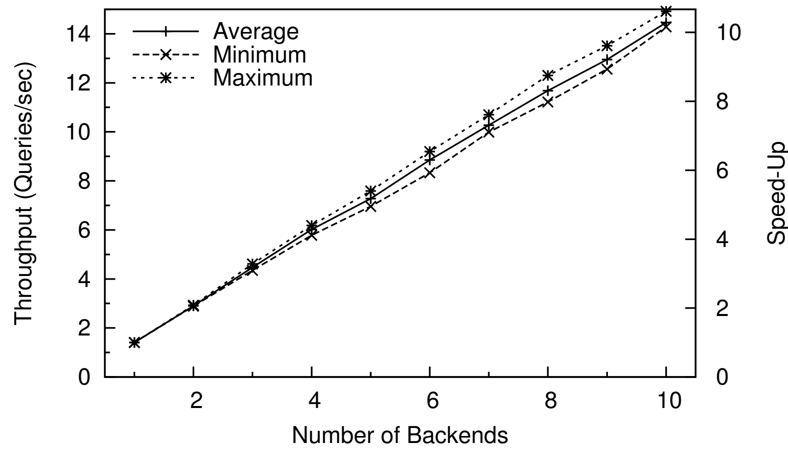



Figure 15.4.: Deviation of the Throughput of the Column Based Allocation

Furthermore, table based and column based allocation reduce the amount of replication substantially. In figure 15.5 the degree of replication of the TPC-H database for one to ten backends can be seen.

The degree of replication $r$ for an allocation $\mathcal{B}$ is calculated as follows:

$$r(\mathcal{B}) = \frac{\sum_{B \in \mathcal{B}} \sum_{f \in B} \text{size}(f)}{\sum_{f \in F} \text{size}(f)} \tag{15.1}$$

Figure 15.5 shows the degree of replication for full replication, table based allocation, column based allocation and optimal column based allocation. Obviously, the degree of replication for full replication matches the number of backends. The table based alloca-

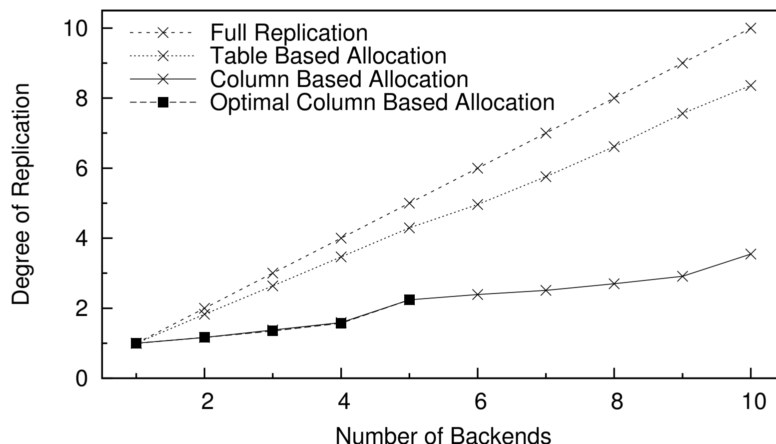



Figure 15.5.: Degree of Replication for Different Cluster Sizes

tion has a slightly reduced degree of replication. The data model in TPC-H is a data warehouse and nearly all queries reference the two biggest tables, which make up 80% of the data. Therefore, it is not surprising that the table based allocation uses over 80% of disk space compared to the full replication. Since the main tables in TPC-H are very broad, column based allocation leads to a considerable reduction of replication. For ten backends, the degree of replication is only 3.5. For reference the result of the optimal allocation, computed by the linear program formulation is shown. Because of the high number of variables and constraints, the optimal allocation could only be calculated for up to 7 backends. It can be seen that the heuristically computed allocation is very close to the optimal allocation. For seven backends the difference in the degree of replication is 0.03. A nice side effect of the reduced degree of replication is an increased allocation speed. Figure 15.6 shows the duration of the allocation procedure for full replication and column based allocation.

The allocation time consists of the preparation of the table fragments, the network transfer time and the data loading. Obviously, the reduced network transfer time and data loading time for column based allocation result from the reduced degree of replication. As shown in figure 15.6, this outweighs the initial overhead of table fragmentation.

## 15.2. TPC-App

To test the update considering allocation we used the TPC-App benchmark. TPC-App is a simulation of an online bookseller which is implemented using web services. A detailed description can be found in section 17.2. The benchmark is scaled by increasing and decreasing the number of customers $EB$. We used $EB = 300$, which resulted in a database size of 280MB. To test the allocation, the workflow of the web services was reimplemented and the queries were automatically generated. The implementation slightly differed from

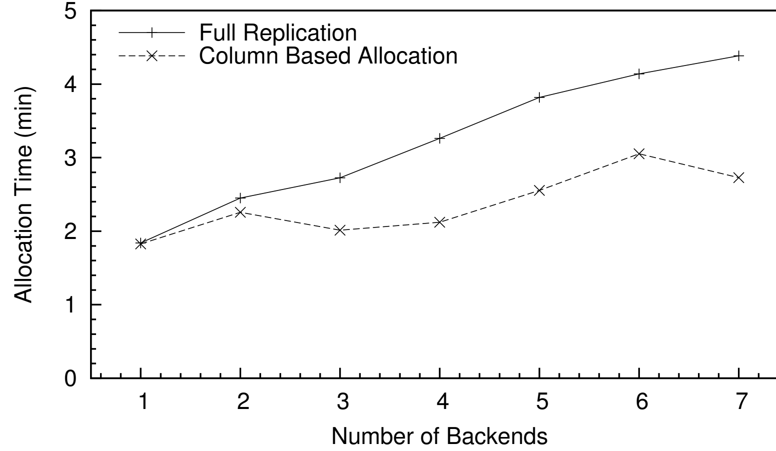



Figure 15.6.: Duration of the Allocation for Different Cluster Sizes

the implementation used in section 6.4; it generated a higher percentage of writes. In the test the number of queries was around 200,000 which we tested on full replication, table based partitioning and column based partitioning. We tested the allocation on 1 to 10 backends, each test was repeated 10 times. The ratio of read to write queries was about 1 to 7. So for each read request 7 inserts and updates were sent. However, the select statements produced overall 3 times more workload than the updates. In particular one complex read query class generated 50% of the workload although its queries made up only 1.5% of all queries. As mentioned above a *read-once/write-all* strategy was used for the query scheduling. The workload consisted of 8 query classes for table based allocation and 10 query classes for column based allocation. All tables that are queried were also updated, therefore the column based allocation always allocated the complete tables.

In figure 15.7 the average speed up of all three allocation strategies can be seen. Due to the high write ratio, the full replication only reaches a speedup of 2.5 which is then stable. Additional backends do not reduce this speedup, but they also do not increase the throughput. Since the weight of the write query classes is 25% in total, this is not surprising. Using the formula presented in equation 3.14 the maximum theoretical speedup can be estimated:

$$speedup = \frac{1}{\frac{parallel}{\#backends} + serial} = \frac{1}{\frac{0.75}{10} + 0.25} = 3.07 \quad (15.2)$$

The maximum speedup achieved by the full replication is 2.6 which is close to the maximum speedup. The table-based and the column-based allocation have a similar speedup that is not limited for the 10 backends. The speedup for the table based and column based allocation can be calculate in relation to the maximum weight of a write request class. In our implementation the writes to the Order_Line table generate about 13% of the query weight, therefore the maximum speedup can be reached if this write

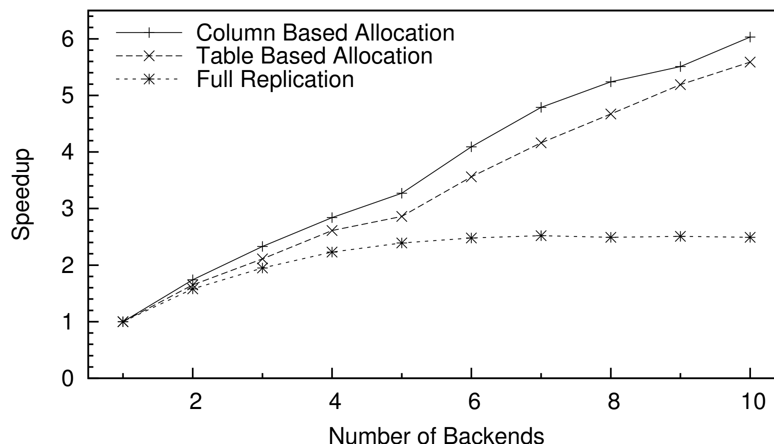



Figure 15.7.: TPC-APP Speedup for Different Cluster Sizes

request class is allocated exclusively on a backend. For 10 backends this results in an increase of the *scale* factor to 1.3. Using equation 12.19 we can compute the theoretical maximum throughput:

$$speedup = \frac{|B|}{scale} = \frac{10}{1.3} = 7.7 \tag{15.3}$$

In our the tests the maximum achieved speedup was 5.8 for table based allocation and 6.7 for the column based allocation. Both are close to the theoretical maximum speedup. In both cases the Order_Line table was allocated on a single backend. The column based allocation achieves a better speedup since it has a more query classes and thus has a more fine grained allocation. In the column based allocation, the backend with the Order_Line table was the bottleneck of the system, while in the table based allocation other backends had a higher load. In the figure 15.8 the total throughput of all configurations can be seen

The throughput has a similar development as the speedup. However, the column based allocation is slightly slower than the table based allocation and the full replication. This is due to some overhead in the query processing for the column based allocation in our implementation. Another effect that can be seen is that for some cluster sizes the average throughput is slightly worse. This can also be seen in figure 15.7, for 5 backends and 9 backends the speedup is slightly decreased. This shows that in some cases the allocation algorithm does not perform as well as in others. These are some corner cases where the algorithm tries to balance the load by allocating small parts of a query class to a backend which then result in an additional write overhead. Since the column based allocation has more query classes it is more vulnerable to these misplacements. This can also be seen in figure 15.9, it shows the deviation of the column based allocation. In contrast to the read only allocation, as shown in figure 15.4, the read write allocation has a higher deviation

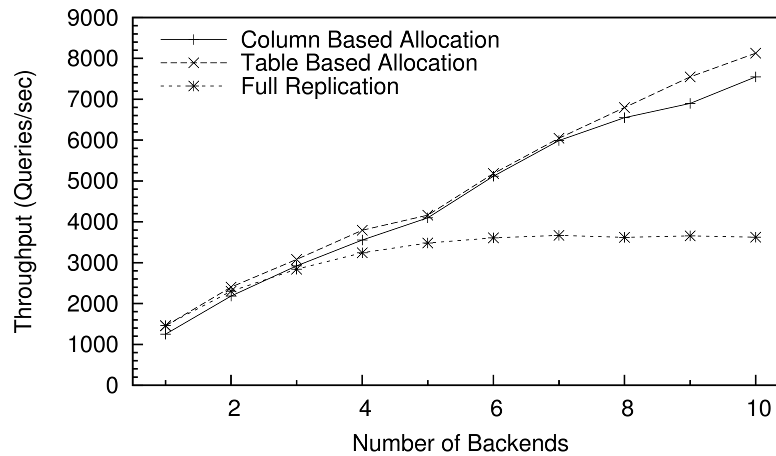



Figure 15.8.: TPC-App Throughput for Different Cluster Sizes

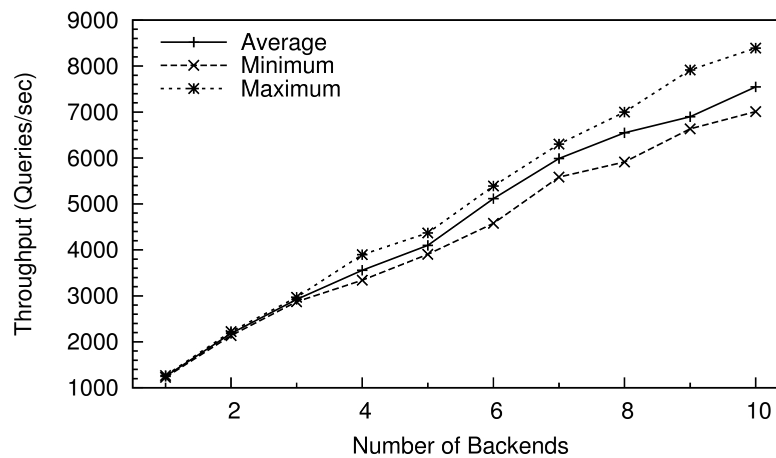



Figure 15.9.: Deviation of the Column Based Allocation

of up to 15%.

## 15.3. Discussion

In this section we have presented two CDBS prototypes and an evaluation of our allocation algorithms. Our first prototype was a fully functional CDBS. Since it had no support for partitioning we implemented a second more basic prototype that allowed us to experiment with partitioning.

We ran two test series on our prototype: first we tested the read only allocation using the TPC-H benchmark, then we tested the read write allocation using the TPC-App

benchmark. The TPC-H tests showed that our algorithm is able to reduce the replication to a factor of 3.5 for 10 backends with column based replication compared to full replication. Furthermore, due to the better I/O performance and cache hit ratio our column based algorithm achieves a super linear speedup of up to 10.6 for 10 backends. The results indicate that the approach is valid and that the approximations are sufficiently accurate.

The test series with the TPC-App benchmark contained a diverse, write intense workload. This is a challenge for the approach since it does not take the effects of diverse query processing times into account. However, the results show a performance that is close to the theoretical optimum. For 10 backends the tests show a speedup of 5.8 for table based allocation and a speedup of 6.7 for column based allocation, this is close to the maximum possible speedup of 7.7 for both configurations. Furthermore, both allocation strategies achieves a throughput that is considerably higher than the theoretical maximum throughput of a fully replicated system.

# 16. Summary

In this part of the thesis we have presented a complete approach for data allocation in cluster database systems. It is designed for the CDBS processing model which was introduced in section 3.4. In the CDBS model every query is executed completely on a single backend, distributed joins and data shipping are avoided. Therefore, the processing time is independent of the network performance. This reduces the complexity of the model to a high degree and makes it possible that all influencing factors can be determined automatically.

The allocation strategy consists of four interrelated steps: query classification, allocation calculation, allocation optimization, and physical allocation. In the query classification step queries are grouped according to the data fragments they access. The granularity of these data fragments determines the partitioning of the data in the allocation step. We have presented methods for relation based classification where the accessed relations determine the query class, attribute based classification where the accessed columns within a relation determine the query class, and predicate based classification which uses the queries predicates to determine the query class. In the literature there are also approaches that classify queries on a per record basis (e.g. [80]), however, in normal sized databases this approach is not feasible. For every query in each query class a weight is determined; we have used the processing time of the queries, which our results proofed to be very accurate measure. Using the weights the relative weight of the query classes can be determined. The goal of the allocation algorithm is to distribute this weight on the backends in such a way that all backends have an equal workload.

We have presented two different allocation problems. One that does not consider updates and therefore tries to minimize the storage requirements and one that considers updates and tries to minimize the additional workload resulting from replicated updates. For both problems we gave a formal definition, an optimal solution using linear programming, a proof of their NP-hardness and a greedy heuristic. Since the greedy heuristics usually produce a non-optimal solution and the complexity of the allocation makes a optimal solution impossible, we also presented a hybrid meta heuristic for both cases that improves the greedy solution. The allocation algorithm is unaware of the method of classification and therefore can be used with any reasonable classification. Since distributed systems tend to be error-prone we also have presented an extension of the algorithm that ensures a tolerance to $k$ failures by replicating query classes at least $k + 1$ times.

In order to implement the calculated allocation in the CDBS an optimal matching is used. We use the Kuhn-Munkres algorithm, it calculates a cost-optimal matching in cubic time. We evaluated both allocation algorithms with standard industry benchmarks. For the read only allocation we have used TPC-H; we have compared a fully replicated system

to a table base allocation and a attribute based allocation. All three allocations scaled linearly, however, the partially replicated solutions achieved a considerable reduction of disk usage, which was up to 65%. Furthermore, the column based allocation actually had a super linear speedup due to the improved cache hit rate and the reduced I/O load.

We have evaluated the update considering allocation using TPC-App an OLTP style benchmark that simulates a web service based online bookseller. The TPC-App workload consisted of a high percentage of updates. Therefore, the full replication of the database limited the speedup to 2.6. The column and table based allocations achieved a much higher throughput of 6.7 and 5.8 for 10 backends. This shows that our approach presents a considerable improvement over a full replication and scales well. Due to the CDBS processing model it can be completely automated and therefore be used for autonomic adaption of database systems.

# Part IV.

# Benchmarking

# 17. Benchmarks

In parts II and III we presented automatic and autonomic approaches for the adaptation and optimization of cluster database systems. Since the complexity of database administration is growing, nearly all major database vendors offer offline database design advisors [232, 82, 12] and recent research considers the online tuning of database systems [50, 227]. Certainly the query workload is the most important variable for physical tuning during runtime. New developments in database benchmarks start to acknowledge this trend. For example, TPC-DS [161] features a new query generator that enables a generation a large set of queries which are syntactically different but semantically similar [179]. Synthetic query streams are usually homogeneous in the frequency of queries and the ratio between different query types, but real database workloads tend to be bursty [130]. New approaches start to acknowledge this fact and define workload as a sequence [13] or chain [127] of statements. This offers new opportunities to adapt the database system to dynamic environments. Nevertheless, there is only little research on how to analyze the efficiency of such approaches. To the best of our knowledge, there is only one publication that introduces a benchmark for autonomic database tuning [73], yet this benchmark also only features homogeneous workloads.

In this section we will introduce new techniques in database system benchmarking that enable realistic workloads to be generated. Furthermore, we will present an approach to massively parallel data generation. The part is organized as follows, in this chapter we will discuss benchmarks used to evaluate the approaches presented in this thesis. Then, we will present our approach to parallel data generation in section 18.1 and the dynamic workload generation in section 18.2. This section of the thesis is ongoing work, which is currently adapted for standard benchmarks in close collaboration with the Transaction Performance Processing Council[1] (TPC).

In order to simulate real life applications of database systems and to test their behavior, benchmark suites were developed. These usually comprise a database, a set of queries and specifications on how to set the queries off. While early approaches had a rather simple layout [44] and the results on different systems were often incomparable, modern benchmarks model realistic business applications and have precise rules that allow accurate comparisons. Today the standard benchmarks for database online transaction processing and decision support systems are developed by the TPC. These have been proved to provide a good representation of industrial workloads [130]. For the evaluation of our approaches, we use the TPC Benchmark$^{TM}$H and the TPC Benchmark$^{TM}$App.

[1]The Transaction Performance Processing Council – `http://www.tpc.org` (last visited 2011-04-15)

## 17.1. TPC Benchmark$^{TM}$H

The TPC Benchmark$^{TM}$H (TPC-H) is a decision support benchmark [183]. It models a data warehouse of an international operating business company. The data model is generic in order to allow a variety of modeled fields of business. TPC-H is a pure OLAP benchmark; it defines a set of complex queries that are executed ad-hoc, i.e. the structure is not known in advance.

In figure 17.1 the schema of the benchmark can be seen. It consists of 8 tables that are interrelated by multiple foreign key constraints. Core of the schema are the Order and Lineitem tables; these represent orders and single positions on the orders, respectively. These two tables serve as fact tables of the schema. This is contrary to the common star schema for data warehouses and is one of the major points of criticism [165]. Together Lineitem and Orders contain over 80% of the data. The size of the data is scaled according to a scaling factor, where scaling factor 100 results in a 100 gigabyte database. Legal scaling factors are 1, 10, 30, 100, 300, 1000, 3000, 10000.

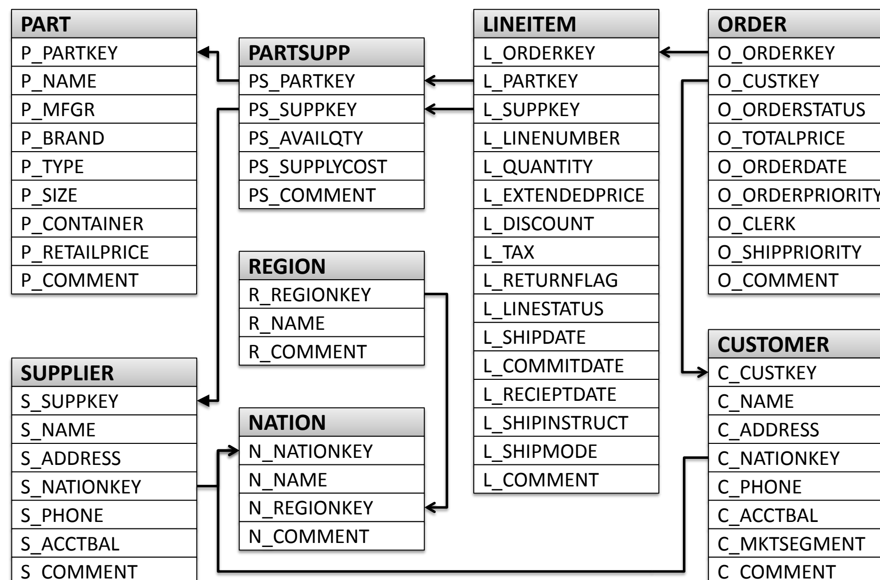



Figure 17.1.: Schema of the TPC Benchmark$^{TM}$H

The benchmark defines 22 complex read-only query templates and two refresh functions. Each of the 22 specified queries models a typical task in business analysis. A relatively simple example is query 6, which quantifies the loss of revenue due to certain company wide discounts:

Listing 17.1: Query 6 of the TPC-H Benchmark

```
SELECT SUM(l_extendedprice*l_discount) AS revenue
  FROM lineitem
```

```
WHERE l_shipdate >= DATE '[DATE]'
  AND l_shipdate < DATE '[DATE]' + INTERVAL '1' YEAR
  AND l_discount BETWEEN [DISCOUNT] - 0.01 AND [DISCOUNT] + 0.01
  AND l_quantity < [QUANTITY];
```

The parameters DATE, DISCOUNT and QUANTITY are randomly replaced within certain domains. The refresh function inserts and deletes a number of rows in the tables Lineitem and Orders, thus keeping the table size constant. The purpose of the refresh function is mainly to ensure the ACID compliance of the system under test.

The performance of a database system is measured in *Queries per Hour* (QphH). Two possible test configurations are defined: a power test and a throughput test. In the power test a single user environment is simulated, hence first the inserts are executed, then the 22 queries and finally the deletes. The duration of the stream measured determines the processing power of a system. In the throughput test multiple query streams, each consisting of the 22 queries, are issued in parallel. An additional stream sends the consecutive refresh functions. The resulting duration determines the throughput of a system. TPC-H also defines a composite metric that is based on the geometric mean of throughput and processing power.

The benchmark aims at a comparable evaluation of the performance of commercial database systems. Therefore, it has a very strict procedure. However, in scientific research various modifications are used. Apart from deviations from the benchmark procedure also modifications of the schema and queries are common. An example is the Star Schema Benchmark by O'Neil et al. [165]. Because of the complexity of the queries, many open source systems have problems processing them[2]. For example, in a performance comparison the developers of MonetDB found out that PostgreSQL produces erroneous results for several TPC-H queries. Therefore, these queries are often adapted. Since the benchmark mainly tests the read throughput, the refresh functions are often omitted. Furthermore, the power test and throughput test are relatively short, hence making the system under test vulnerable to initialization overheads. Therefore, multiple consecutive query streams are frequently used.

## 17.2. TPC Benchmark$^{TM}$App

The TPC Benchmark$^{TM}$App (TPC-App) is primarily an application and web services benchmark [100]. It is the successor of the TPC Benchmark$^{TM}$W (TPC-W), which was designed as a transactional web benchmark. Because of the low acceptance of both benchmarks, they were discontinued by the TPC and marked as obsolete. TPC-App is an online transaction processing benchmark that simulates application servers and web services that perform business-to-business (B2B) transactions. The purpose of the benchmark was to measure the performance of commercial application server products.

[2]Confer the performance comparison of MonetDB, MySQL and PostgreSQL - `http://www.monetdb-xquery.org/SQL/Benchmark/TPCH/` (last visited 2011-06-26)

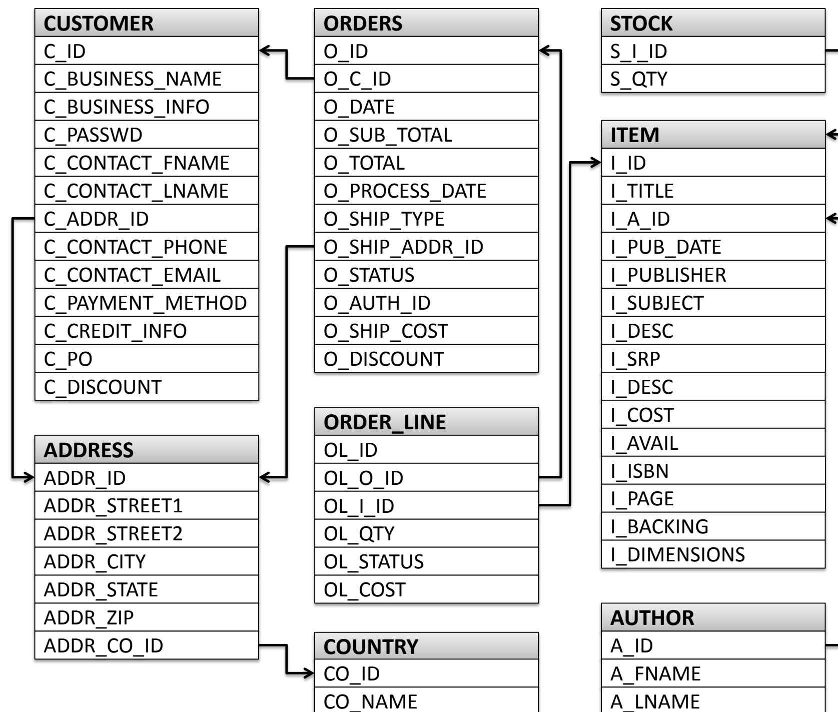



Figure 17.2.: Schema of the TPC Benchmark$^{TM}$App

For database performance measurements the benchmark is particularly interesting since it models a web environment, which is a major application area of database systems.

In figure 17.2 the schema of the TPC-App benchmark can be seen. It consists of 8 tables that model a B2B book store. The tables are scaled according to the emulated business clients (EB). For each EB 192 customers in the Customer table are generated. The tables Address, Orders and Orderline are scaled according to the scale of Customer. The Item and Stock tables are fixed to 100,000 and the Author table is fixed to 25,000. For 100 EBs the database has a size of 176 megabytes.

The benchmark defines 7 different web services that are processed as transactions on the database system. The web services are invoked by so-called active emulated business clients (active EB). Each of the clients issues one transaction at a time. The transactions and their relative probability can be seen in the table below:

| Web Service | Percentage |
|---|---|
| New Customer | 1% |
| Change Payment Method | 5% |
| Create Order | 50% |
| Order Status | 5% |
| New Products | 7% |
| Product Detail | 30% |
| Change Item | 2% |

The benchmark has a predefined maximum processing time per transaction. In order to scale the throughput of the benchmark, the number of active EBs has to be scaled. The performance of the system is then measured in the number of *Web Service Interactions per Second* (SIPS).

Although both TPC-W and TPC-App are obsolete, they are frequently used in academic projects to measure the performance of web related database systems. These systems rarely use the complete benchmark specification. The web service architecture is only of limited interest for the database system performance. Therefore, the benchmarks are used like OLTP benchmarks. We chose the TPC-App benchmark for our tests, since it was still active when the first tests were conducted.

Apart from the performance metrics, TPC also specifies two additional metrics for pricing and energy efficiency. These can be applied to all the TPC benchmarks. The pricing metric compares the performance relative to the price of the system, i.e. the hardware and software costs and maintenance for 3 years. The result is the price per performance unit, e.g. *Price per Queries per Hour* (Price/QphH). The energy efficiency metric measures the energy consumption per performance [181], e.g. *Watt per Queries per Hour* (Watt/QphH).

A shortcoming of the TPC benchmarks is that they are static in their database access behavior, while real life systems often face shifting workloads. To simulate this for the adaptive approaches, we use log data from an e-learning management system.

## 17.3. E-Learning Benchmark

In order to find a possibility to simulate realistic database workloads for the TPC-App benchmark, real life online information systems are a promising application domain. Usually it is very hard to get any detailed information about the structure and especially the workload of such systems, since they are treated as industrial secrets. However, we are in the fortunate position of having access to a sufficiently large online e-learning management platform that is used at the University of Passau. It was chosen as a basis of our benchmarking workloads.

Stud.IP[3] is a popular e-learning management system. It started as a simple forum and evolved into a full-featured Course and Campus Management System over the years. The

[3] Stud.IP - `http://www.studip.de` (last visited 2011-04-15)

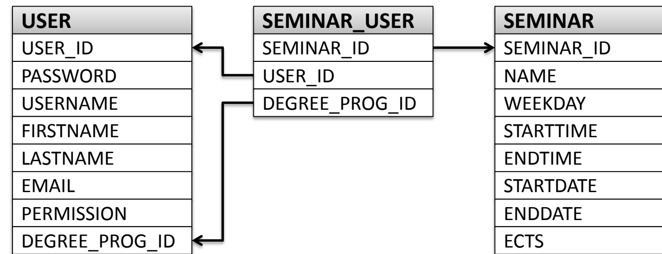



Figure 17.3.: Excerpt of the E-Learning Benchmark Schema

system supports the complete course life cycle, beginning with creating the course, filling it with data, assigning times and rooms, specifying application procedures and exporting the data into PDF or HTML. Online communication and cooperation are encouraged by providing a forum for each course, wiki, messaging system, chat and online material. Today, 38 universities and 17 other institutes are using Stud.IP[4], one of them is the University of Passau.

Stud.IP is written in PHP and uses a MySQL database. On a normal day during the semester, between 50 and 100 parallel users are online at any given time. At the beginning of a new semester, this number is drastically higher, normally there are about 200-300 users online at the same time. The normal MySQL load is at about 1,200 database requests per second as each PHP page generates several database requests.

In the spring semester of 2009, there were 1,734 courses with a total of 15,047 registered users of which 1,374 had a teacher role. Among those users, 672 teachers and 7,072 students logged in at least once during the semester. 6,921 of those student users are registered in courses with a total of 63,895 course registrations. Since the launch of Stud.IP in fall 2006, 8,907 courses have been entered, 222,349 course registrations processed, 52,017 documents uploaded and 178,070 internal messages sent. The database has 7,688,642 entries and is 1.3 GB in size.

We had the opportunity to get log traces of the Apache web server and an anonymized dump of the database. As described in section 6.4 the log traces were used to simulate varying workloads for the TPC-App benchmarks. However, we also used the data to specify a complete benchmark that was presented in [187].

The database of the e-learning benchmark is only a fraction of the Stud.IP schema as it is used at the University of Passau. For reasons of simplicity it has been reduced to the core functionality, thus it only consists of 25 tables compared to the 200 tables in the production system. In figure 17.3 the main tables of the course management can be seen. The table User stores information about the users, which are students, teaching staff or employees. The table Seminar contains all courses and lectures and their properties. The relation between users and seminars is established via the table user_seminar. Obviously, there are many more tables that further specify the course management and other parts

[4]`http://www.studip.de/referenzen/` (last visited 2011-04-15)

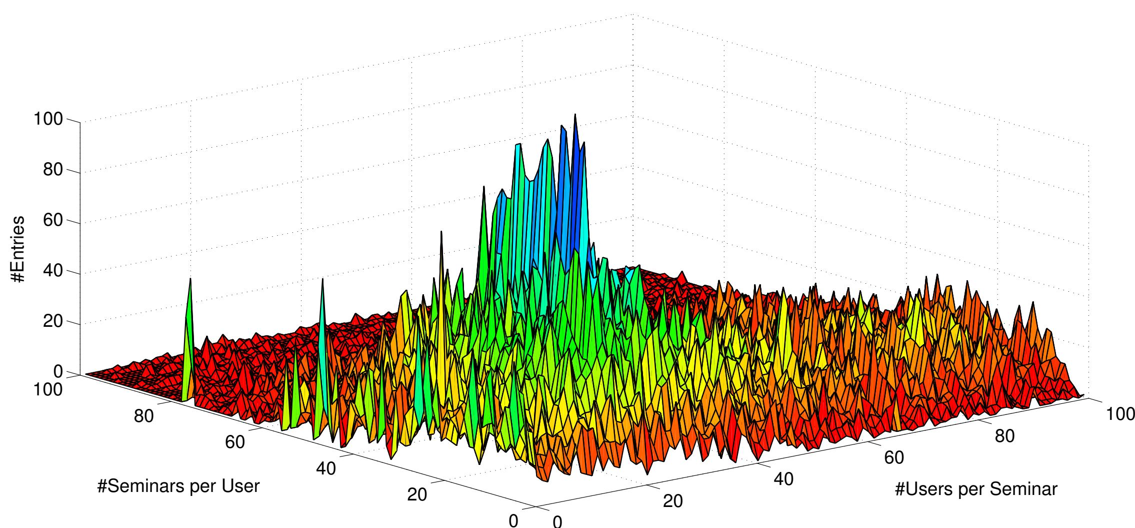



Figure 17.4.: The Reference Distribution in the Table Seminar_User

of the system.

To populate the schema above, we have analyzed the value and reference distributions between the tables. For the table `seminar_users` a reference distribution can be seen in figure 17.4. In order to allow a synthetic generation of the reference distribution, we used maximum likelihood estimation to fit standard probability distributions to the data. For this data we used lognormal distributions, since they model most distributions sufficiently (for a discussion about lognormal distribution see [157]). Figure 17.5 shows that the distribution of the number of seminars a user is registered for can be modeled by a lognormal distribution, even though a gamma distribution would produce a better fit. The distribution of the number of users per seminar does not match the log-normal distribution very well, but for generation purposes it is still sufficient. This can be seen in figure 17.6. Similar observations about reference distributions were made by Hsu et al. in [130]. They used the Hill equation to model the references, which is related to the loglogistic distribution.

Since we only had the Apache log traces, we had to extract the queries from the system. We used the log traces to find the websites that were accessed most frequently. In figure 17.7 the top 5 most accessed websites in June 2008 can be seen. Based on these accesses, we analyzed the relevant PHP scripts and formulated SQL queries that reflect the generated database accesses based on our schema. An example can be seen in listing 17.2. This query retrieves information about a user that will be presented at the login page.

Listing 17.2: SQL Query from the E-Learning Benchmark

```
SELECT s.name, u.vorname, u.nachname, ui.address, ui.phone, u.email
```

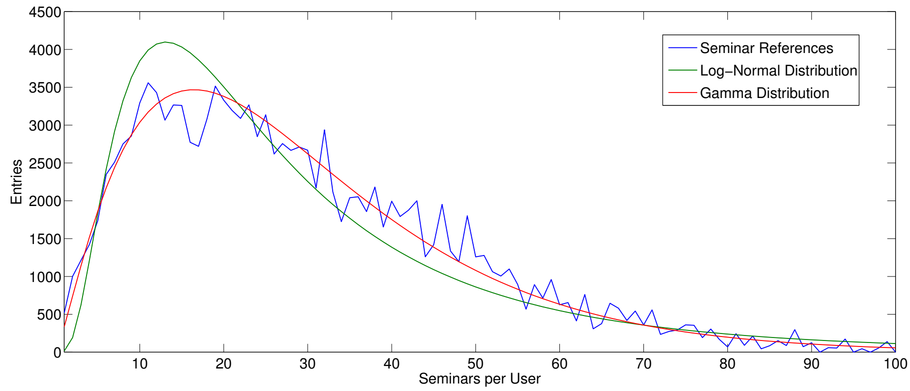



Figure 17.5.: Distribution of Seminars per User in Table Seminar_User

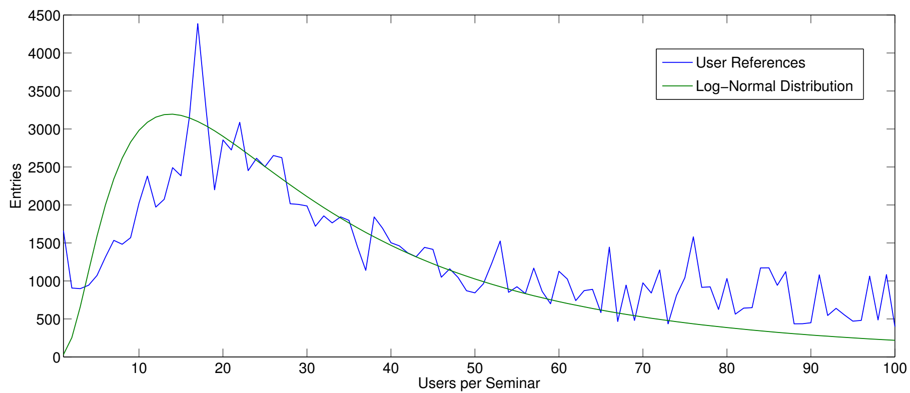



Figure 17.6.: Distribution of Users per Seminar in Table Seminar_User

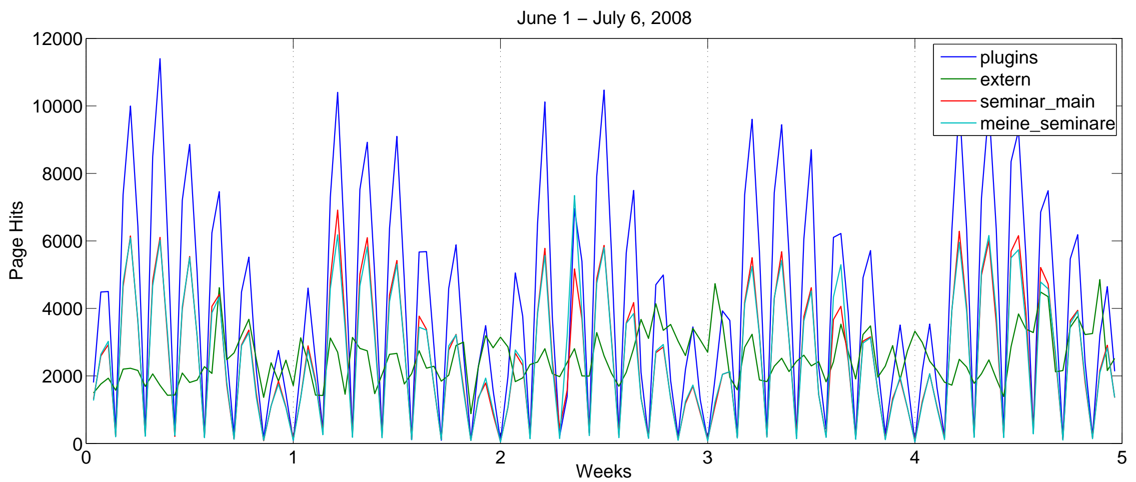



Figure 17.7.: Most Accessed Websites in June 2008 per 6 Hours

```
FROM user u, user_info ui, user_degree_prog ud, degree_program d
WHERE u.user_id = ui.user_id
  AND ud.user_id = u.user_id
  AND ud.degree_prog_id = d.degree_prog_id
  AND u.user_id = '[USER]';
```

It can be seen that the log traces show daily and weekly patterns. Each stream has a peak around noon each day. This is typical for any system with user interaction, the same patterns can for example be found in the Wikimedia logs in section 4.3. Therefore, we used time series analysis to allow an accurate representation of the workload. We will give details of the analysis and the model in the next chapter.

# 18. Benchmarking Large Dynamic Systems

In the previous chapter, we presented two standard database system performance benchmarks. These benchmarks test the performance of database systems by measuring peak performance on homogeneous request streams. Nevertheless, in systems with user interaction access patterns are constantly shifting as was shown in the analysis of the e-learning system above. Even though database access in most cases is triggered by human interaction, programs generate the actual SQL code. Therefore most queries are very similar and can be divided into relatively few distinct classes. Within these classes only simple parameters, like predicates change. Due to user interaction the occurrence of the classes depends on timetables. The most important examples are the day and night rhythm and the weekly cycle. In figure 17.7 in section 17.3 this can be seen clearly for the accesses of Stud.IP system. It is easy to see that there is a daily and a weekly period. Each of the website accesses displayed will generate at least one and in most cases a sequence of SQL queries. For one website the queries will only differ in the variables. Apart from the difference in the amount of workload between day and night and workday and weekend, shifts in the workload between the single classes can also be seen. In figure 18.1 an average of the days in the data above is pictured. Not all websites are accessed in the same pattern. Thus, depending on the time of day, the database will have different access rates and different access patterns.

Similar access patterns can be seen for any user accessed information system, see for example the access rates at the Wikimedia clusters shown in figure 4.2 in section 4.3. This periodic behavior provides opportunities for optimization. On the one hand peak loads get more predictable and in times of low access the database can be prepared for the higher load. However, there is little work on how to generate workloads which comprise such patterns. Most benchmarks have static workloads in terms of content and amount. As mentioned before this applies, for example, to the TPC benchmarks. Like TPC benchmarks, most benchmarks measure only the peak performance of a system. However, since many systems are notoriously underloaded this is not realistic [134]. For many applications such as energy efficiency and scaling variable loads are required. A recent approach by Beitch et al. tries to generate more realistic workloads by simulating concurrent users [31]. Although the simulation of users makes very complex workloads possible, it lacks of repeatability. Due to the parallel random generation of user accesses results can vary even on a single machine. Furthermore, the simulation of single users does not automatically generate daily and weekly patterns, Beitch et al. propose to increase the number of simulated users one by one. A similar approach is used by SPECpower[1], which

[1]SPECpower website - `http://www.spec.org/power_ssj2008/` (last visisted 2011-06-26)

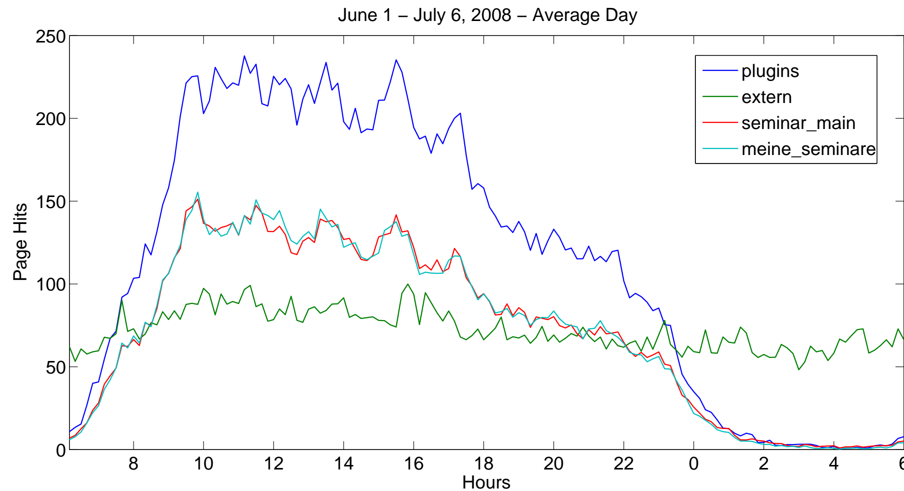



Figure 18.1.: Most Accessed Websites in June 2008, Average Day per 10 Minutes

increases the load in 10% steps. To generate more realistic workloads we present a time series based model. This model generates workloads based on real workload distributions.

Apart from varying workloads, there is a continuous growth of data sizes which is already beyond petabyte scale for many applications. This poses new challenges for the research community. Usually, data of this size is stored in large clusters or clouds. Processing large data sets demands a higher degree of automation and adaptability than smaller data sets. However, traditional benchmarks are not sufficient for cloud computing, since they fall short on testing cloud specific challenges [41]. Currently, there are only a few benchmarks available specifically for cloud computing. The first one was probably the TeraSort Benchmark[2]. Others followed, such as MalStone, CloudStone [204], and YCSB [75]. These benchmarks are dedicated to a single common task in cloud computing. While this kind of benchmarks is essential for scientific research and evaluation, it fails to give a holistic view of the system under test. Hence, there is a need for a benchmark suite that covers various aspects of large dynamic systems such as clouds. The database community traditionally has an affection for simple benchmarks [108, 209]. Although reduction of complexity is a basic principle of computer science and unnecessary complexity should be avoided at all costs, there seems to be a trend to simplistic evaluations. In order to get meaningful results benchmarks should have diverse and relevant workloads and data [45]. It is often best to use real life data. For example, in scientific database research the Sloan Digital Sky Survey is frequently referenced [215]. Yet for many applications, such as social websites, there is no data publicly available. And even though storage prices are dropping rapidly, they are still very high for petabyte scale systems. Therefore, it is not sensible to store petabytes of data only for testing purposes. Besides storage, the network necessary to move petabytes of data in a time efficient manner is costly. Hence, the data should be

[2]The current version can be found at `http://www.sortbenchmark.org/` (last visited 2011-04-15)

created where it is needed. For a cluster of nodes this means that each node generates the data it will process later, e.g. load into a data base. In order to generate realistic data, references have to be considered, which usually requires reading already generated data. Examples of references are foreign keys. For clusters of computers this results in a fair amount of communication between nodes. In order to generate data on cloud scale systems efficiently, the generation has to be cloud-aware. That means it has to pursue the top goals of cloud computing, namely scalability and decoupling, i.e. avoidance of any interaction of nodes [42].

In the following, we will present our approach to parallel generation of large scale relational data, and after that a new model for realistic workload generation.

## 18.1. Data Generation

Data generation for performance evaluation is part of the daily business of researchers and DB administrators. Most of their data generators are special purpose implementations for a single dataset. There has been quite a lot of research on data generation for performance benchmarking purposes. An important milestone was the paper by Gray et al. [110], the authors showed how to generate data sets with different distributions and dense unique sequences in linear time and in parallel. Fast, parallel generation of data with special distribution characteristics is the foundation of our data generation approach.

According to their reference generation procedure, data generators can be roughly divided into three categories: no reference generation, scanning references, and computing references. No reference generation means that no relationships between tables are explicitly considered. So references are either only simple or based on mathematical probabilities. In this scheme it is, for example, not possible to generate foreign keys on a non-continuous unique key. Examples are data sets that only consists of single tables or data sets (e.g. SetQuery [166], TeraSort, MalGen [34], YCSB [75]) or unrelated tables (e.g. Wisconsin database [44], Bristlecone[3]).

Scanned references are generated reading the referenced tuple. This is either done simultaneously to the generation of the referenced tuple or by scanning the referenced table. This approach is very flexible, since it allows a broad range of dependencies between tables. However, the generation of dependent tables always requires the scanning or calculation of the referenced table. When the referenced table is read, additional I/Os are generated, which in many applications will limit the maximum data generation speed. Generating tables simultaneously does not constitute a problem. However, it requires generating all referenced tables. This is very inefficient if the referenced tables are very large and do not need to be generated, e.g. for a materialized view with aggregation. Most systems that generate references use scanned references. An example is dbgen[4], the data generator provided by the TPC for the TPC-H benchmark[180]. Another approach

[3]Available at `https://bristlecone.svn.sourceforge.net/svnroot/bristlecone/trunk/bristlecone/` (last visited 2011-06-26)

[4]dbgen can be downloaded from `http://www.tpc.org/tpch/` (last visited 2011-06-26

was presented by Bruno and Chaudhuri [49]: it largely relies on scanning a given database to generate various distributions and interdependencies. Houkjær et al. describe a graph-based generation tool that models dependencies in a graph and uses a depth-first traversal to generate dependent tables [129]. A similar approach was presented by Lin et al. [147]. Two further tools that offer quite similar capabilities are MUDD [206] and PSDG [124]. Both feature description languages for the definition of the data layout and advanced distributions. Furthermore, both tools allow parallel generation. However, as described above the independent generation of dependent data sets is not possible.

A computed reference is recalculated using the fact that the referenced data is deterministically generated. This results in a very flexible approach that also makes it possible to generate data with cyclic dependencies. The downside is the computational cost for regenerating the keys. To the best of our knowledge, our approach is the only one that relies on this technique for parallel data generation.

### 18.1.1. Parallel Random Number Generation

The basis of synthetic data generation is the use of random number generation. In most cases deterministic data generation is necessary to allow repeatable experiments on various kinds of systems. Therefore, pseudo random number generators are used, which allow deterministic generation of random looking data. Many different kinds of pseudo random number generators have been proposed. They differ highly in the statistical quality of their data and the length of their period. The quality of random numbers is important to avoid the creation of patterns. Equally important for generating very large data sets is the period length of the pseudo random number stream. Every generator eventually repeats the sequence of its data. It therefore has to be assured that the period is large enough for the amount of data to be generated.

In order to generate data for large data sets it is necessary to generate data in parallel, for which parallel pseudo random number generators have been developed. In contrast to linear generators, parallel generators do not reseed the data generator after every number. A linear generator ($lrng$) calculates a random number essentially as follows:

$$rng(n) = \underbrace{lrng(lrng(\dots lrng(seed)\dots))}_{n \text{ times}} \tag{18.1}$$

Efficient implementation of linear pseudo random number generators store the last generated value as their internal state. Parallel random number generators ($prng$), however, are often stateless by generating random numbers independently using a hash function:

$$rng(n) = prng(seed + n) \tag{18.2}$$

Examples can be found in [184]. Using this approach, the generation of random number sequences can easily be distributed to many parallel processes. In order to generate the same sequence on different numbers of processors, either the leapfrog method or sequence splitting is used [94, 71]. While the leapfrog method partitions the sequence between

processes in turn, sequence splitting partitions the sequence in contiguous sequences. The concrete value generation is a function from the random number to a value. An example of a generator of realistic data is a name generator based on a dictionary lookup. To generate a name two entries of a list of common first and last names can be picked based on a random number. Usually, random number generators generate uniformly distributed values. However more natural distributions can be calculated easily. For normal distributions the Box-Muller method and the related, usually faster, polar method can be used [150].

### 18.1.2. Deterministic Data Generation

Based on the parallel pseudo random number generation realistic data with dependencies can be generated. In order to achieve an acceptable generation speed, the determinism in the random number generation can be exploited. As explained above, each single independent value in the database can be generated by a function that maps the random number to a concrete value. To generate dependent values, the generation can be based on either the same random number or on the initial value.

Consider the excerpt of the Stud.IP schema presented in figure 17.3. There are three tables, *User*, *Seminar*, and *Seminar_User*. The generation of tables *user* and *seminar* are straightforward. For *seminar_user* only *User_id*s and *Seminar_id*s must be generated that actually exist in *User* and *Seminar*. This is only easy if both attributes have continuous values, otherwise it is necessary to check that the referenced tuples exist. A second challenge is that *Degree_Prog_ID* is replicated in *Seminar_User*, so the combination of *user_id* and *degree_program* have to exist in *user*. Finally the values in *seminar_user* have a non-uniform distribution as shown in figure 17.4.

The common solution to generate the table *seminar_user* is to first generate the two tables that are referenced and then use a look-up or scan to generate the distribution. If this is done in parallel, either the referenced data has to be replicated, or the data generating process has to communicate with other nodes. This is feasible for smaller clusters, but for cloud scale configurations the communication will produce a bottleneck.

Therefore, we propose a fully computational approach. Basically, our data generation is a set of functions that map a virtual row id to a tuple's attribute. Using this approach, we can easily recompute every value. So for the example above we would define a function for each attribute in the original tables. To generate uncorrelated data, the first computational step is usually either a permutation or a pseudo random number generation.

For the example above this is only needed for the *degree_program*. The value can either be chosen from a dictionary or be generated. To generate entries of *seminar_user*, two pseudo random numbers in the range of $[1, |user|]$ and $[1, |seminar|]$ are computed with given distribution properties. Then the function to generate *degree_program* is used, resulting in a valid tuple for *seminar_user*. This can be computed independently of the generation of *user* and *seminar*. Since parallel pseudo random number generators are used, *seminar_user* can be generated on any reasonable number of computing nodes in parallel.

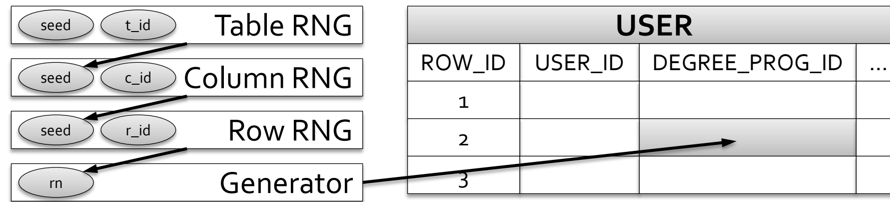



Figure 18.2.: Hierarchical Seeding Strategy

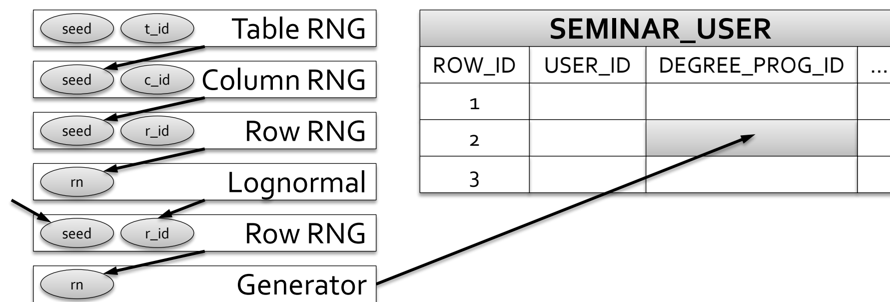



Figure 18.3.: Hierarchical Seeding Strategy for References

This flexibility opens up a broad field of application. Besides traditional relational, row oriented data, our system can easily generate data in other storage models, such as the Decomposed Storage Model [77], column wise as in MonetDB [46] or C-Store [211] or even mixed models [190].

To generate field values independently and to avoid costly disk reads, a random number generator and a corresponding seed are assigned to each field. The random number generators used can be organized hierarchically, so that a deterministic seed for each field can be calculated. This can be seen in figure 18.2. To generate the random number of a certain field in the database the seed for the relevant column has to be calculated. Starting from a single seed for the complete database (project) a seed for every table is generated. With this seed a new random number generator is seeded and used to generate seeds for every column. With the column seed the column random number generator is seeded, which generates random numbers for every field in the database. These random numbers are deterministically mapped to a value.

As the number of tables and columns are static, their seeds can be cached after the first generation. Hence, it is not necessary to run through the complete seeding hierarchy to determine the seed for a column. It is sufficient to re-seed the column random number generator with its precalculated seed, skip forward to the row needed and get the next value. Using a hash based random number generator this is a very inexpensive operation. After seeding, the random number is passed to the corresponding field generator to generate the actual value.

To ensure referential integrity most data generators either have severe restrictions on the

key generation or read generated keys to generate consistent data. Using the deterministic approach, referential integrity can be ensured by regenerating valid keys. The row number of each value is used as a surrogate key. To generate a valid key, the random sequence of the key column and the key generator is needed. By picking a random row of the key column the relevant key can be generated. In order to generate statistical distributions of references the random picking of row numbers can be distributed appropriately. This is depicted in Figure 18.3, to generate a referenced value in the first step a random lognormal distributed value within the number of rows of the referenced table is generated. This value is used to regenerate the key value. Again, it is not necessary to process the complete seeding hierarchy, since it is possible to cache all occurring seeds.

Obviously, it is also possible to generate other dependencies than simple foreign key constraints. The basic idea to solve intra row dependencies is to use different streams of random numbers for unrelated data and equal streams for related data. If two columns must contain related data, for example different formatting of the same data, they must share the same random sequence. Obviously, an attribute can depend on multiple other attributes. In order to apply the approach above, the generation of the dependent value has to consider either the random numbers of all values it depends on or the generated values.

### 18.1.3. Implementation

In order to evaluate the performance of the data generation approach, we implemented it as a generic, extensible framework. The parallel data generation framework (PDGF) was written in Java with a focus on platform independence. It has a plug-in architecture that allows an easy extension, which can be seen in figure 18.4.

**Controller/View** The controller takes user input such as the configuration files from the command line or a built-in interactive mini-shell. This input is used to configure an instance of PDGF and to control the data generation process. By default PDGF is a stand alone command line tool, but there is also a graphical user interface. The controller allows the use of PDGF as a library. Distributed startup is currently implemented by an external shell script.

**Random Number Generator** A parallel random number generator is the key to make the seeding algorithm efficient and fast. PDGF's random number generator is based on a hash function as explained above. The default random number generator can also be exchanged and it is also possible to specify a custom random number generator for each generator.

**Generators** The generators determine how the values for a field are generated. PDGF comes with a set of generators for general purpose: ids, numeric values, references, dictionary based, pseudo text grammar and several special generators for names, etc. Since

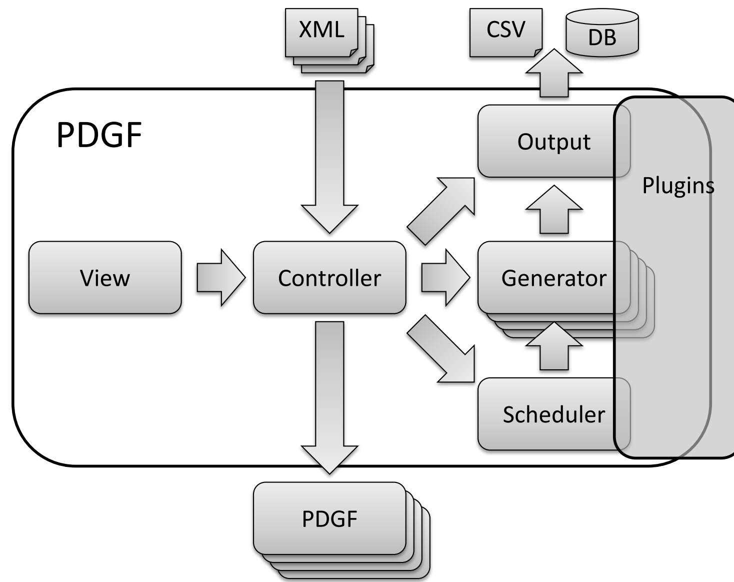



Figure 18.4.: Architecture of the Parallel Data Generation Framework

some data sets require a special structure, e.g. the TPC-H benchmark data set, PDGF provides a simple interface enabling easy implementation of generator plug-ins.

**Distribution Functions** Distribution functions allow generators to adapt and exchange the statistical distributions of generated values. The distribution functions transform the uniform random numbers provided by the random number generator into non-uniformly distributed values. As for the generators, PDGF comes with basic distribution functions: beta, binomial, exponential, log-normal, normal, Poisson, and Student's-t.

**Output Module** The output module determines how to save the generated data. An output module receives the values of an entire row for a table along with some meta information. By default the generated data is written to a comma separated value file, one per table and instance. Another possibility is to convert the generated values into SQL insert statements. These can either be written to a file or sent directly to a DBMS.

**Scheduler** The scheduler calculates the work of the PDGF instance and assigns it to the threads on the node. PDGF's default strategy is to statically divide the work between nodes and workers in chunks of equal size. This is efficient if the data is generated in a homogeneous cluster or similar environment. In a heterogeneous environment the static approach leads to varying elapsed times among the participating nodes.

**Configuration** PDGF is configured by several XML-based configuration files. One file configures the runtime environment, while the others configures the data model and gen-

eration routines. The runtime configuration is optional and can also be specified by command line parameters. It specifies which part of the data is generated on a node and how many threads are used. This information is used by the scheduler for splitting the work between participating nodes. Listing 18.1 shows a sample runtime configuration file. This example is for Node 5 out of 10 nodes. Two worker threads are used.

Listing 18.1: "Runtime Configuration File"

```
<nodeConfig>
 <nodeNumber>5</nodeNumber>
 <nodeCount>10</nodeCount>
 <workers>2</workers>
</nodeConfig>
```

The model configuration file is used to specify the data model. It follows a hierarchical structure that reflects a relational database schema. In figure 18.2 an excerpt of the configuration of the Stud.IP example can be seen. The schema definition begins with default definitions such as the scaling factor, the seed, and the random number generator. After that the tables are specified. Each table has a size which can be specified in relation to a scaling factor. For each field the type is given and a generator, which can be further specified.

Listing 18.2: "Data Model File Example for Seminar_User References."

```
<schema name="simpleELearning">
 <scaleFactor name="user">10000</scaleFactor>
 <seed>1234567890</seed>
 <rng name="PDGFDefaultRandom" />
 <tables>
  <table name="user">
   <size>user</size>
   <fields>
    <field name="user_id">
     <type>java.sql.Types.INTEGER</type>
     <generator name="IdGenerator">
      <unique />
     </generator>
    </field>
    <field name="degree_program">
     <type>java.sql.Types.VARCHAR</type>
     <size>20</size>
     <generator name="DictList">
      <file>dicts/degree.dict</file>
     </generator>
    </field>
   </fields>
  </table>
  <table name="seminar">  [..]  </table>
  <table name="seminar_user">
```

```
   <size>20 * user</size>
   <fields>
    <field name="user_id">
     <type>java.sql.Types.INTEGER</type>
     <reference>
      <referencedField>user_id</referencedField>
      <referencedTable>user</referencedTable>
     </reference>
     <generator name="DefaultReferenceGenerator">
      <distribution name="LogNormal">
       <mu>7.60021</mu>
       <sigma>1.40058</sigma>
      </distribution>
     </generator>
    </field>
    <field name="degree_program"> [..] </field>
    <field name="seminar_id">      [..] </field>
   </fields>
  </table>
 </tables>
</schema>
```

To enable an output that differs from the data model a second file is used to specify the actual generation. This is necessary if only part of the data model has to be generated. Furthermore, it allows the data model to be separated from the generation specification. An example of a generation specification can be seen in figure 18.3. It contains the scheduler, the output module and its configuration and a list of all tables that have to be generated. In order to allow different forms of output for different tables, the output can be defined for every table.

Listing 18.3: "Data Model File Example for Seminar_User References."

```
<project name="simpleELearning">
 <scheduler name="FixedJunkScheduler"/>
 <output name="CSVRowOutput">
  <outputDir/>
  <fileEnding>.txt</fileEnding>
  <delimiter>|</delimiter>
  <bufferSize/>
 </output>
 <schema name="simpleELearning">
  <tables>
   <table name="user">
    <output name="CSVRowOutput">
     <outputFile>User\_File</outputFile>
    </output>
   </table>
    [..]
  </tables>
```

```
  </schema>
</project>
```

### 18.1.4. Performance

We used the TPC-H and the SetQuery databases to evaluate the performance of the data generation approach. All tests were conducted on a high performance cluster with 16 nodes. Each node had two Intel Xeon QuadCore processors with 2 GHz clock rate, 16 gigabyte RAM and two 74 GB SATA hard disks configured with RAID 0. Additionally, a master node was used, which had the same configuration, but an additional hard disk array with a capacity of 2 TBytes. For both benchmarks two test series were conducted. The first series tested the data generator's scalability in terms of data size on one node. The second series demonstrated the data generator's scalability in terms of the number of nodes with fixed data size. Each test was repeated at least five times. All results are averages of these test runs.

**SetQuery** The SetQuery data set consists of a single table *BENCH* with 21 columns [166]. 13 of the columns are integers with varying cardinalities from 2 to $1,000,000$ of which 12 are generated randomly. 8 additional columns contain strings, one with 8 characters and the others with 20 characters. The table size is scaled linearly according to a scaling factor $SF$, where $SF = 1$ results in about 220 MB. First we generated a 100 GB data set on 1 to 16 nodes (i.e. scaling factor 460). As can be seen in Figure 18.5 the average speed up per node is linear to the number of nodes. One node was able to generate about 105 MB per second. The super linear speed up for a higher number of nodes results from caching effects, which can be seen more clearly in the second test.

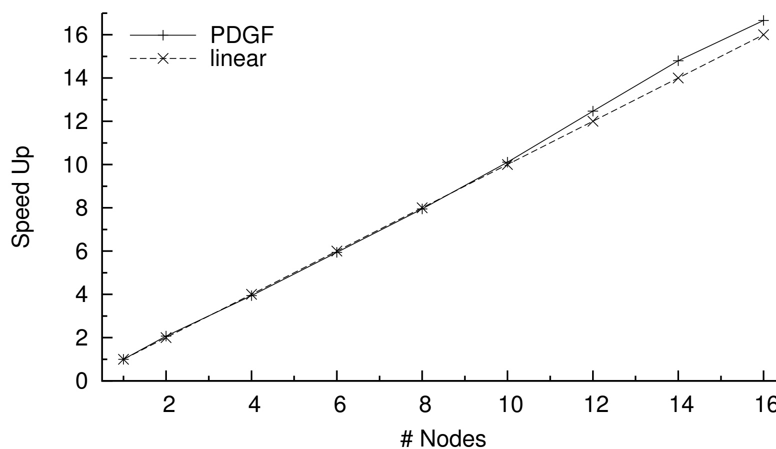



Figure 18.5.: Scaleup Results for 1 to 16 Nodes for a 100 GB SetQuery Data Set

For the second test, different data sizes were generated on a single node. We used

scale factor 1 to 460. The resulting durations are shown in figure 18.6. It can be seen that the data generation scales well with the amount of data. However, the generation is not constant. This is due to caching effects and initialization. For smaller data sizes the initialization overhead decreases the overall generation speed. Then at scaling factor 10 (i.e. about 2 GB) there is a peak that results from hard disk and disk array caches. For larger data sizes the hard disks write speed produces a bottleneck and limits the generation speed to about 100 MB/s.

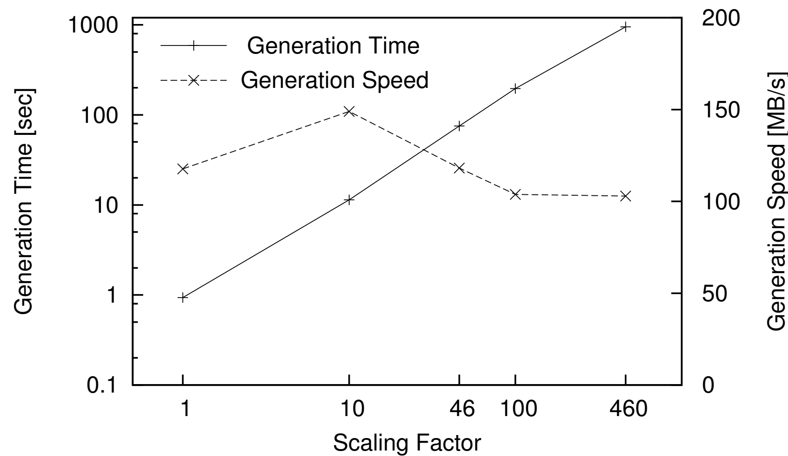



Figure 18.6.: Generation Time and Speed for Different Scaling Factors of the SetQuery Data Set

**TPC-H** To test a more complex scenario and compare the generation speed with other data generation tools, we used the data generation approach to generate TPC-H data. Again, we tested the data generator's scalability in terms of the amount of data and the number of nodes. Figure 18.7 shows the data generation elapsed times for scale factor 1, 10 and 100 for a single node. Additionally, we generated the same data sizes with dbgen. Both axes of the figure are in logarithmic scale. To obtain fair results, dbgen was started with 8 processes, thus fully exploiting the 8 core system. Generation times for both tools were CPU bound. Since we had notable variations in the runtime of dbgen, we only report the best of 5 runs for each scaling factor. As can be seen in the figure, our configurable and extensible Java implemented tool can compete with a specialized C implementation.

Figure 18.8 shows the data generation times for different cluster sizes. For all cluster sizes the data generation duration scales linearly with the data size. Furthermore, the generation time for certain scale factors decreases linearly with the number of nodes it is generated on. However, for scale factor 1 on 10 and 16 nodes the generation speed is significantly slower than for the other configurations. This is due to the large initialization overhead compared to the short generation time.

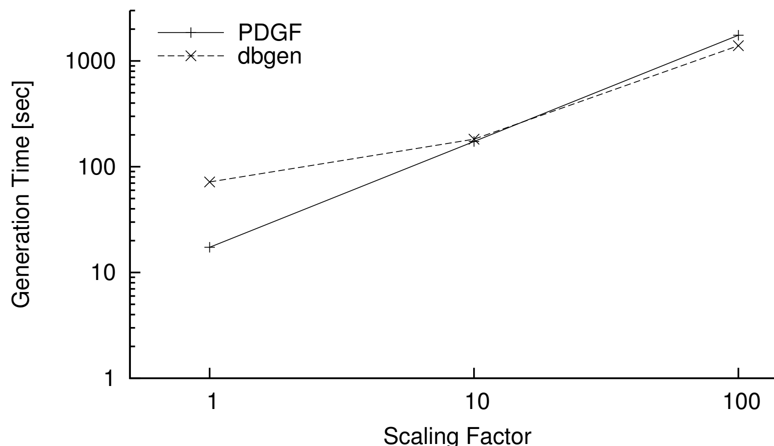



Figure 18.7.: Comparison of the Generation Speed of dbgen and PDGF

## 18.2. Workload Generation

To benchmark adaptability in database systems, the query generator has to be able to simulate realistic workloads that shift in quantity and ratio of the requests. Therefore, the design focus was to build a realistic workload model that reflects user dependent workload patterns. For that purpose we proposed a new kind of random generator for time series in [187].

The basis of the generator is the assumption that the workload comprises several components that can be modeled independently and that these components can be represented by an approximating polynomial. This model was previously applied by Calzarossa and Serazzi to classify and characterize workload patterns [53]. As was shown for the workloads of the Stud.IP system and the Wikimedia clusters, the workload consists of daily patterns, hence we use our model to represent the workload of a single day. Since the single queries in the workload can be classified, we use a single representation for each class. The workload is not continuous (see figure 18.1), therefore, it is smoothed. We use an aggregation of 60 minutes. Hence, the workload of a single class is a time series consisting of 24 measurements, each reflecting the number of accesses in a time interval of 60 minutes. The time series starts at 4 am in the morning, when the number of accesses is at the low point (see figure 18.1).

For each day and each class we are given a time series consisting of $N + 1 = 24$ observations $y_n$ at time points $x_n$, with $n \in \{0, \dots, N\}$. These points are equidistant in time. Therefore, we can find an optimally approximating polynomial of degree $K$ using a linear combination of $K + 1$ basis polynomials $p_k$:

$$p_{\boldsymbol{a}}(x) = \sum_{k=0}^{K} a_k p_k(x) \tag{18.3}$$

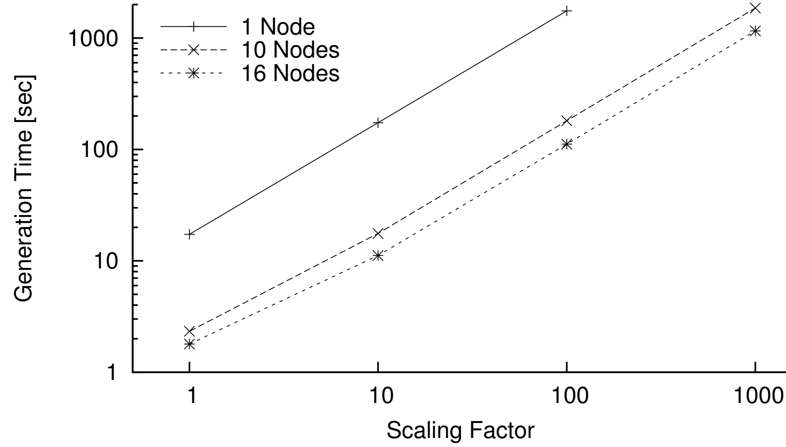



Figure 18.8.: Generation Times of TPC-H Data Sets on Different Cluster Sizes

where $a_k \in \mathbb{R}$ are the weighting factors of the basis polynomials and build the weighting vector $\boldsymbol{a}$. An optimal approximation can be found using the least squares approximation. Calzarossa and Serazzi use monomials to approximate the time series. Hence, they found the weighting factors to be clustered but not following a standard distribution. However, we require the polynomials to have the following properties:

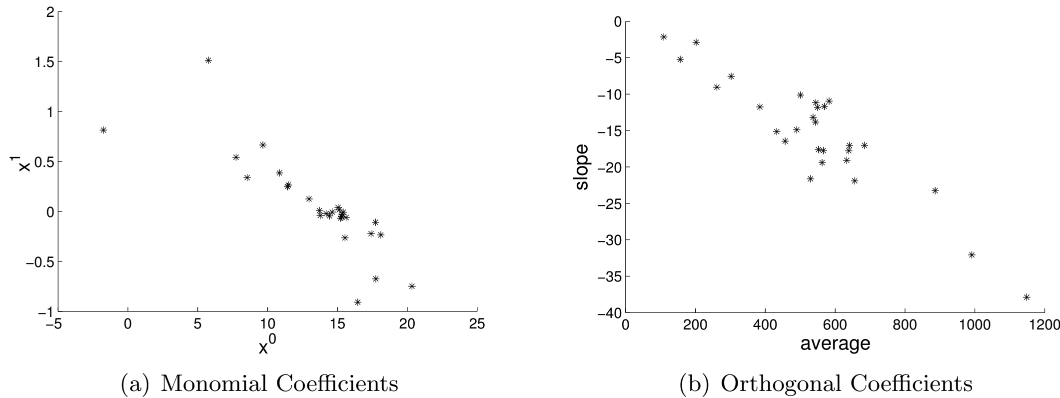
- Their degree must ascend from 0 to $K$.
- The leading coefficient (coefficient of the monomial with the highest degree) of each basis polynomial must be one.
- Each pair of basis polynomials $p_{k_1}$ and $p_{k_2}$ (with $k_1 \neq k_2$) must be *orthogonal* with respect to the inner product

$$\langle p_{k_1} | p_{k_2} \rangle = \sum_{n=0}^{N} p_{k_1}(x_n) p_{k_2}(x_n). \tag{18.4}$$

  That is, $\langle p_{k_1} | p_{k_2} \rangle = 0$ for all $k_1 \neq k_2$.

Since the observations were made at equidistant points in time, the choice of the basis polynomials depends only on their number $N + 1$. We assume that the first observation is made at time 0, otherwise we simply shift the time series to this point. In the context of orthogonal basis polynomials, the weighting factors $a_k$ are called *orthogonal expansion coefficients*. These are optimal estimators of the *average* ($a_0$), *slope* ($a_1$), *curve* ($a_2$), *change of curve* ($a_3$), etc. of the time series [96, 97]. The mathematical background can be found in [88, 95].

When using the orthogonal base polynomials for the approximation it can be seen that the representations of the sample time series all originating from a particular kind of




(a) Monomial Coefficients (b) Orthogonal Coefficients

Figure 18.9.: Distribution of Monomial and Orthogonal Coefficients for Degree 0 and 1

day (e.g., public holiday, working Friday, etc.) can be regarded as being nearly *normally distributed*. This can be seen in figure 18.9; here the coefficients of the approximation of the most visited website of the Stud.IP system on Mondays in the semester in winter term 2008 can be seen. It can be seen that in contrast to monomial coefficients (see figure 18.9(a)), the orthogonal coefficients (see figure 18.9(b)) can be modeled by a *multivariate Gaussian distribution*:

$$\mathcal{N}(\boldsymbol{a}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{(K+1)/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{a}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\boldsymbol{a}-\boldsymbol{\mu})\right) \tag{18.5}$$

where $\boldsymbol{\mu}$ is the $(K+1)$-dimensional center (or mean) and $\boldsymbol{\Sigma}$ the $(K+1) \times (K+1)$-dimensional covariance matrix. To find $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ a standard maximum likelihood technique can be applied [43].

Based on these findings, a model for time series that represent a workload of a certain working day can be built by approximating a set of sample time series (ideally more than $N$) as described above and computing the resulting multivariate Gaussian distribution. Using the model a random time series can be generated in the following way:

1. Using a random number generator points within the multivariate Gaussian distribution can be generated.
2. These points can be transformed into the respective polynomials using the (known) orthogonal basis polynomials.
3. The polynomials can be evaluated at the desired points in time, resulting in the according intensity of the workload.
4. If a more unsteady workload is needed random noise can be added, for example white noise with a standard deviation corresponding to the average approximation error for the set of sample time series.

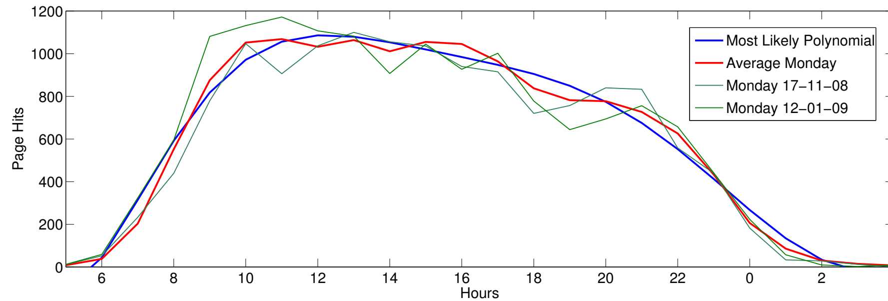



Figure 18.10.: The Most Likely Approximating Polynomial for Mondays During the Lecture Period

Using this approach, a generator for an arbitrary large set of artificial workloads can be built, which all have a similar shape to the original time series. An example of a polynomial approximation for Mondays during the semester can be seen in figure 18.10; the average workload on Mondays as well as the most likely approximating polynomial can be seen; the orthogonal coefficients of the most likely polynomial specify the mean of the multivariate distribution. For comparison two regular workloads are shown.

The approximating polynomials tend to $\pm\infty$ in the limit case. This can result in negative values at the boundaries of the modeled interval. In order to avoid this problem, the interval analyzed can be extended at both ends. The complete procedure is the same, except that the borders are not evaluated for the workload generation.

As shown before, each day of the week has different access rates. Therefore, we built single models for every day of the week. In this way we can also easily simulate holidays and outliers with anomalous accesses.

### 18.2.1. Scaling Time

An important factor for the usability of a benchmark is its runtime [45]. The smallest unit of time that has periodical access rates is usually one day. To test adaptability several periods have to be processed. Since this is too long for most benchmarking purposes, we scale time (see section 6.4). With a scaling factor of 1/7 a complete week can be simulated within 24 hours. Depending on the application under test, even smaller factors could reasonable be tested. Another possibility to shorten runtime is to use a reduced week that only consists of three days.

Of course, the system under test has to be aware of the time scaling factor. Since daily and weekly periods are usual in information systems, good tuning processes will use this knowledge for periodical tasks.

## 18.3. Benchmarking Objectives

Depending on the benchmark objective, different test cases can be built. Shifting workloads give lots of opportunities to test automatic and autonomic systems. For database systems a common metric is *transactions per second* or *average response time* for a given database size, depending on the optimization goal (e.g. the *QphDS@SF* metric in TPC-DS [161]). It has to be mentioned that whichever is used, the other should also be monitored (for example TPC benchmarks define an upper limit for the response time). In the following we will give four examples of how shifting workloads can improve benchmarking.

### 18.3.1. Basic Performance

The most common benchmarking objective in database systems is to test the speed, i.e. transactions per second or similar. A good baseline for such a test is the peak performance of the system without any automatic tuning and without any workload shifts.

To test if the system can automatically produce a better throughput in a real life environment, alternating workloads can be used. This way the system has phases of high load, which can be used to measure the peak performance. In phases of low load, the system has time to optimize its table structure, scale itself or tune the indices without risking serious performance bottlenecks. Throughout the test the ratio of different query classes stay constant. After some periods the peak performance should increase and should be better than the baseline performance.

### 18.3.2. Adaptability

As stated before a major goal was to measure adaptability. The idea is to test how well a system can adjust itself to the workload. As we have shown before, the rates of query classes change within a single day. This can be simulated by shifting workloads. So different query sets are defined and for each set a separate time series is generated. Also the workload is different for each day of the week. Either a complete week can be simulated, or a reduced week consisting of only two working days and one weekend day, which should suffice in most cases. With this test, a system under test that is aware of the temporal dependencies in the workload should get a better performance than a system that is not.

Changes in the workload behavior can be introduced to further test the adaptability. In figure 18.11 the most frequently accessed websites in Stud.IP between October 08 and May 09 can be seen. It is easy to see that there are sections with very different characteristics. The diagram starts shortly after the beginning of the semester, which lasted until the first week of February. The next semester started on April 20. Additionally the Christmas break from December 24 until January 06 can be seen. So for an eLearning system at a university a week can be classified in one of the three classes, semester, semester break and holidays. All three of these sections are well-defined and their ranges are previous knowledge. This form of test is in some respects already implemented in current

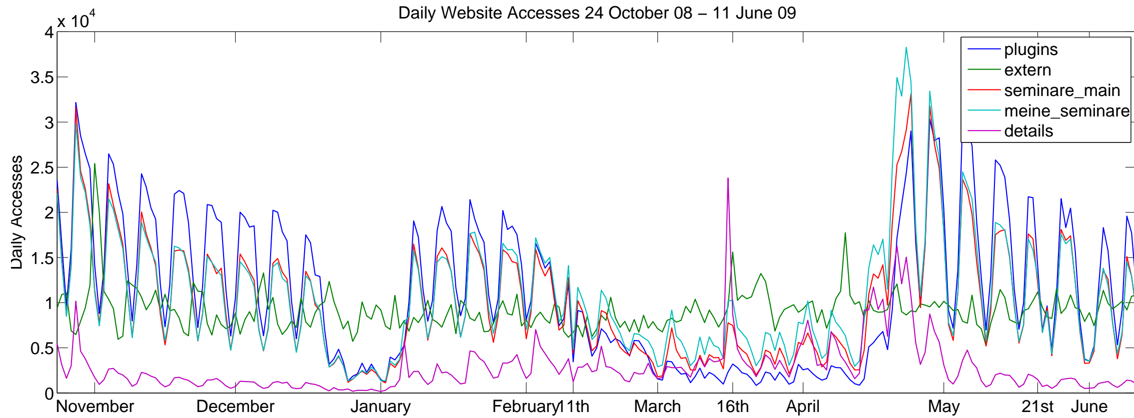



Figure 18.11.: Most Accessed Websites in Stud.IP Between October 24, 2008 and June 10, 2009 per Day

benchmarks, TPC-DS, for example, consists of four consecutive phases with very different characteristics (i.e. load, query run, data maintenance, query run - cf. [182]). However, our form of query generation also makes it possible to model the trends within one phase. Such a trend can be seen in the fall term 2008 when the workload constantly decreased and then slightly increased at the end of the term.

### 18.3.3. Robustness

To test the robustness of an autonomic system outliers can be introduced. In figure 18.11 these can be seen in form of legal holidays on May 21 and June 1 and in the form of unpredictable outliers for example on February 11 (server maintenance) or March 16 (unexpected user behavior). An autonomic system should be able to identify such outliers and deal with them correctly. So, it should not change its configuration completely based on that single day. Yet, it also must not have a serious performance collapse. For national holidays this could also be presented by previous knowledge. Outliers can be modeled like other days and either triggered randomly (maintenance) or at previously defined points in time. To test robustness the performance before and after an outlier can be compared and the time until the original performance level is reached again. To find out if a system is *over adapted*, the performance during an outlier day can be used.

## 18.4. Discussion

In this part we have presented new techniques for benchmarking. First, we have explained our approach for the deterministic generation of realistic data sets. After this we have shown a new model to generate arbitrary workloads that simulate realistic user behavior. Our contributions are the following:

- The deterministic seeding approach enables the generation of realistic data sets.
- Using parallel random number generators our generic data generation approach has equivalent speed as a native C implementation.
- Based on the deterministic generation complex dependencies are possible.
- The time series model for workload generation enables the generation of realistic workloads.
- Based on a real e-learning application a benchmark for dynamic workloads was specified.

Obviously, the data generation approach and the workload model can be combined. This makes workloads possible that have non uniform access patterns. This enables, for example, a simulation of hot spots in the database. On the other hand, integrating the workload model in the data generation makes it possible to generate data which has time based dependencies, such as timestamps.

# Part V.

# Conclusion

# 19. Conclusion

In this thesis, we have presented a processing model for the cluster database architecture, a common architecture for distributed database systems. The model allows an accurate analysis of the requirements of a distributed database system. Many research projects use very complex models to get an exact image of the query processing in the distributed system. These models, however, limit the application to small problem sizes or very controlled environments. Our model allows an automatic computation of all parameters. Therefore, all algorithms based on this model can be automatized. This increases the possibilities for self-management.

Based on this model we presented an autonomic approach for the scaling of cluster database systems. The scaling was implemented in the generic framework Scalileo. Scalileo features an online feedback control loop based on the MAPE model. It can easily be integrated into existing applications to extend them with self-management and particularly with autonomic scaling. We have shown two show cases as a proof of concept: a distributed web server and a cluster database system. Both applications were scaled according to their workload. Apart from the reduced management costs for installation and integration of additional nodes, the systems have a substantially increased energy efficiency. For the distributed web server with 4 nodes a reduction of the energy consumption by 30% was possible and for the cluster database system a reduction of energy consumption by 24% was achieved.

To increase the throughput and scalability of the CDBS, we presented a formal definition of the allocation problem in cluster database systems. As for the processing model, the allocation problem is reduced to the necessary parameters, which can be computed automatically. We have shown optimal and heuristic allocation algorithms that optimize the storage efficiency of the cluster database system architecture. Furthermore, the query processing performance is increased. Two versions of the algorithm were presented: a version for read-only workloads and a version for workloads with updates on the database. Both were evaluated with industry standard benchmarks. The evaluation showed that for read only workloads our algorithm can compute allocations that reduce the storage requirements by 65% and achieve super linear speedup. The allocation algorithm for workloads with updates increases the performance by up to 2.4 times compared to a fully replicated system and achieves a throughput that is not possible for a fully replicated system.

Since there is no benchmark for database systems in dynamic environments, in the final section we presented methods to generate realistic data and realistic workloads in cluster environments. For the data generation we have developed a generic approach that allows massively parallel generation of databases over large numbers of nodes. The resulting

data features complex dependencies and the generation speed has an ideal speedup. The workload generation produces query streams with realistic variations and frequencies. The model is based on an orthogonal polynomial approximation which allows a calendar based classification of the workloads. Since coefficients of polynomials representing workloads of equivalent periods in time have a normal distribution, these distributions can be found with standard techniques. They can be used to generate arbitrary numbers of varying query streams with similar, realistic characteristics.

# 20. Ongoing and Future Work

In this chapter, we will give details of current and future work. We have presented a model for cluster database systems, which has several limitations due to the simple processing model 3.4.1. In future work, we will introduce further features to the model, such as transactions and distributed query processing. This will enable us to adapt our allocation and scaling approaches and increase the area of application of our model.

## 20.1. Scaling

The autonomic scaling framework presented in part II can be extended to support all kinds of autonomous tasks. An important extension will be the introduction of stochastic models for the threshold estimation. Using hidden Markov models or support vector machines will allow an automatic adaption of the parameters for scaling and the like. Since many workloads have a periodic pattern, time series analysis will allow a proactive approach for adaptation. Using the workload model presented in section 18.2 combined with a clustering approach such as k-nearest neighbors, we will be able to classify a workload to a certain group of days and predict the shape of the workload.

## 20.2. Allocation

We have presented several algorithms for the allocation problem in cluster database systems in part III. In future work we will implement the attribute based classification, in order to provide horizontal partitioned data layouts. In our tests we have used a baseline configuration of the backend database systems. However, the performance of database systems can be increased drastically with local schema optimization strategies. Most promising seem dynamic materialized view and index selection, as presented in [50, 199, 149]. These will require the consideration of the local database layout, which we saw as a black box in this work.

Obviously, the allocation strategies can be used for various other applications as well. Using the weighting of the backend system capabilities, the optimal placement of data on heterogeneous storage systems can be optimized as presented in [55].

Furthermore, we will extend our research in allocation strategies for highly dynamic environments. Part of this work will be a more integrated approach of the scaling with the advanced allocation strategies. We are currently also working on a pure online version of the allocation algorithm. This will provide an option to improve the allocation without excessive data transfer.

To better exploit modern hardware architectures, it would be sensible to combine our data parallel approach with local declustering techniques to increase the processing of single requests. This would lead two a two staged architecture, reflecting the shared nothing architecture on cluster level and the shared memory architecture of modern multiprocessor/multi-core systems [121].

## 20.3. Benchmarking

The methodologies presented in part IV allow a wide area of application. We are currently exploring applications for medical informatics; due to data protection laws, it is in general not possible to work with real life data. Generating synthetic medical records will allow researchers to test medical information systems with large, realistic data sets.

The TPC is specifying a new benchmark for ETL systems [228]. Due to the genericness and performance of our data generator, we have been invited to implement the official data generator for this. This data integration benchmark consists of multiple data sources that have to contain the same database in different models. The source systems are represented by an OLAP style database model, an XML file containing user data, a mixed file containing various types of information and a flat file containing news data. The target system is modeled as a data warehouse. All data have to be consistent with each other. Furthermore, random errors in the data have to be generated. This introduces new challenges for the data generation, which we are currently studying and which we approach with generic extensions to our data generation framework.

In order to generate consistent updates and queries, we will integrate the workload generation in the data generation framework. Using the deterministic value generation, various access patterns can be realized. This enables new workload characteristics such as hot spots in data access, without the necessity of simulating single users as, for example, proposed by the Rain workload generator [31]. Including the workload model in the data generation will also enable temporal dependencies in the data. In real data sets, time stamps of orders or the like will generally not be uniformly distributed, but follow a pattern that matches the access frequency. This can be simulated using polynomial representation.

# Bibliography


[1] Information Technology - Database Language SQL, 1992.

[2] Magic Packet Technology. Technical Report 20213, Advanced Micro Devices, 1995.

[3] An Architectural Blueprint for Autonomic Computing. Technical report, IBM Corporation, 2006.

[4] DRDA V4: Distributed Relational Database Architecture (DRDA), 2007.

[5] Oracle Database 11g Performance and Scalability. Whitepaper, Oracle, 2007.

[6] HP Neoview Advantage - A Data Warehouse Platform for the new Generation of Business Intelligence. Technical report, Hewlett-Packard Development Company, 2010.

[7] Oracle Real Application Clusters in Oracle VM Environments. Whitepaper, Oracle, 2010.

[8] The Vertica Analytic Database Technical Overview White Paper. Technical report, Vertica Systems Inc., 2010.

[9] Unified Modeling Language, 2010.

[10] M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. P. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Early Experiences on the Journey Towards Self-* Storage. *IEEE Data Engineering Bulletin*, 29(3):55–62, 2006.

[11] R. Agrawal, S. Chaudhuri, A. Das, and V. R. Narasayya. Automating Layout of Relational Databases. In *ICDE '03: Proceedings of the 19th International Conference on Data Engineering*, pages 607–618, 2003.

[12] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *VDLB '04: Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 1110–1121. Morgan Kaufmann, 2004.

[13] S. Agrawal, E. Chu, and V. Narasayya. Automatic Physical Design Tuning: Workload as a Sequence. In *SIGMOD '06: Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 683–694, New York, NY, USA, 2006. ACM.

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

[15] A. Ailamaki, D. J. DeWitt, M. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, Edinburgh, Scotland, UK, September 1999. Morgan Kaufmann.

[16] S. Akioka and Y. Muraoka. HPC benchmarks on Amazon EC2. In *WAINA '10: Proceedings of the 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, pages 1029–1034, 2010.

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

[18] S. Albers. Onlinealgorithmen - Was ist es wert, die Zukunft zu kennen? *Informatik-Spektrum*, 33(5):438–443, 2010.

[19] M. Allman, K. Christensen, B. Nordman, and V. Paxson. Enabling an Energy-Efficient Future Internet Through Selectively Connected End Systems. In *HotNets '07: Proceedings of the Sixth Workshop on Hot Topics in Networks*, 2007.

[20] P. A. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570. IEEE Computer Society Press, 1976.

[21] A. C. Alvim, C. C. Ribeiro, F. Glover, and D. J. Aloise. A Hybrid Improvement Heuristic for the One-Dimensional Bin Packing Problem. *Journal of Heuristics*, 10(2):205–229, 2004.

[22] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Kogge, J. Levesque, D. Reed, V. Sarkar, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling. ExaScale Software Study: Software Challenges in Extreme Scale Systems. Technical report, Georgia Institute of Technology, 2009.

[23] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS '69: Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, 1967.

[24] R. R. Amossen. Vertical Partitioning of Relational OLTP Databases Using Integer Programming. In *ICDEW '10: 26th IEEE International Conference on Data Engineering Workshops*, pages 93–98, 2010.

[25] E. Anderson and J. Tucek. Efficiency Matters! In *HotStorage '09: Proceedings of the SOSP Workshop on Hot Topics in Storage and File Systems*, New York, NY, USA, 2009. ACM.

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

[27] J. Armstrong. A History of Erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, 2007.

[28] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, and A. M.-S. nd Marco Protasi. *Complexity and Approximation - Combinatorial Optimization Problems and Their Approximability Properties.* Springer Verlag, 2003.

[29] L. A. Barroso, J. Dean, and U. Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

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

[31] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox, and D. A. Patterson. Rain: A Workload Generation Toolkit for Cloud Computing Applications. Technical Report UCB/EECS-2010-14, Electrical Engineering and Computer Sciences – University of California at Berkeley, 2010.

[32] C. Belady. Green Grid Data Center Power Efficiency Metrics: PUE and DCiE. Technical Report 6, The Green Grid, 2008.

[33] K. Bellam, A. Manzanares, X. Ruan, X. Qin, and Y. Yang. Improving Reliability and Energy Efficiency of Disk Systems via Utilization Control. In *ISCC '08: IEEE Symposium on Computers and Communications*, pages 462–467, 2008.

[34] C. Bennett, R. Grossman, and J. Seidman. MalStone: A Benchmark for Data Intensive Computing. Technical report, Open Cloud Consortium, 2009.

[35] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10, San Jose, California, May 1995. ACM, ACM Press.

[36] J. Bernardino and H. Madeira. Experimental Evaluation of a New Distributed Partitioning Technique for Data Warehouses. In *IDEAS '01: Proceedings of the International Database Engineering & Applications Symposium*, pages 312–321, 2001.

[37] P. A. Bernstein, N. Dani, B. Khessib, R. Manne, and D. Shutt. Data Management Issues in Supporting Large-Scale Web Services. *IEEE Data Engineering Bulletin*, 31(4):3–9, 2006.

[38] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2009.

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

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

[41] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the Weather Tomorrow?: Towards a Benchmark for the Cloud. In *DBTest '09: Proceedings of the Second International Workshop on Testing Database Systems*, pages 1–6, New York, NY, USA, 2009. ACM.

[42] K. Birman, G. Chockler, and R. van Renesse. Toward a Cloud Computing Research Agenda. *SIGACT News*, 40(2):68–80, 2009.

[43] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.

[44] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking Database Systems: A Systematic Approach. In *VLDB '83: Proceedings of the 9th International Conference on Very Large Data Bases*, pages 8–19, San Francisco, CA, USA, November 1983. ACM, Morgan Kaufmann Publishers Inc.

[45] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Communications of the ACM*, 51(8):83–89, 2008.

[46] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. In *VLDB '09: Proceedings of the 35th International Conference on Very Large Data Bases*, pages 1648–1653. VLDB Endowment, 2009.

[47] A. B. Bondi. Characteristics of Scalability and Their Impact on Performance. In *WOSP '00: Proceedings of the 2nd International Workshop on Software and Performance*, pages 195–203. ACM, 2000.

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

[49] N. Bruno and S. Chaudhuri. Flexible Database Generators. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1097–1107. VLDB Endowment, 2005.

[50] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE '07: Proceedings of the 23rd International Conference on Data Engineering*, pages 826–835. IEEE, 2007.

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

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

[53] M. Calzarossa and G. Serazzi. A Characterization of the Variation in Time of Workload Arrival Patterns. *IEEE Transactions on Computers*, 34(2):156–162, 1985.

[54] L. Camargos, F. Pedone, and M. Wieloch. Sprint: A Middleware for High-Performance Transaction Processing. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 385–398, 2007.

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

[56] S. Casner and S. Deering. First IETF Internet Audiocast. *ACM SIGCOMM Computer Communication Review*, 22(3):92–97, 1992.

[57] E. Cecchet. RAIDb: Redundant Array of Inexpensive Databases. In *Parallel and Distributed Processing and Applications, Second InternationalSymposium, ISPA 2004*, pages 115–125, Hong Kong, China, December 2004. LNCS 3358, Springer Verlag.

[58] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based Database Replication: The Gaps Between Theory and Practice. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 739–752, 2008.

[59] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *Proc. USENIX Annual Technical Conference, Freenix Track*, Boston, MA, USA, June 2004.

[60] S. Ceri, S. Navahe, and G. Wiederhold. Distribution Design of Logical Database Schemas. *IEEE Transactions on Software Enginieering*, 9(4):487–504, 1983.

[61] S. Ceri and G. Pelagatti. *Distributed Databases - Principles and Systems.* McGraw-Hill, Inc., 1984.

[62] S. A. Ceri, M. Negri, and G. Pelagatti. Horizontal Data Partitioning in Database Design. In *SIGMOD '82: Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 128–136, 1982.

[63] J. S. Chase, D. C. Anderson, P. N. Thakar, A. Vahdat, and R. P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating System Principles*, volume 35 of *ACM SIGOPS Operating Systems Review*, pages 103–116, New York, NY, USA, 2001. ACM.

[64] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao. Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services. In *NSDI '08: 5th USENIX Symposium on Networked Systems Design & Implementation*, pages 337–350. USENIX Association, 2008.

[65] J. Chen, G. Soundararajan, and C. Amza. Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers. In *ICAC '06: Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, 2006.

[66] P. P.-S. Chen. The Entity-Relationship Model — Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[67] W.-J. Chen, R. Ahuja, Y. J. Bi, R. Borovsky, P. Fürer, C. Maddux, I. Ohta, and M. Talens. *DB2 Integrated Cluster Environment Deployment Guide*. 2004.

[68] W. W. Chu. Optimal File Allocation in a Multiple Computer System. *IEEE Transactions on Computers*, 18:885–889, 1969.

[69] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(1):377–387, 1970.

[70] E. F. Codd. *The Relational Model for Database Management: Version 2*. Addison Wesley, 1990.

[71] P. D. Coddington. Random Number Generators for Parallel Computers. *National HPCC Software Exchange Review Electronic Journal*, 2, 1996.

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

[73] M. P. Consens, D. Barbosa, A. M. Teisanu, and L. Mignet. Goals and Benchmarks for Autonomic Configuration Recommenders. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 239–250, New York, NY, USA, 2005. ACM.

[74] Continuent. *Tungsten Concepts and Administration Guide*, 2010.

[75] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, New York, NY, USA, 2010. ACM.

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

[77] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, pages 268–279, New York, NY, USA, 1985. ACM.

[78] D. W. Cornell and P. S. Yu. A Vertical Partitioning Algorithm for Relational Databases. In *ICDE '87: Proceedings of the Third International Conference on Data Engineering*, pages 30–35, 1987.

[79] G. D. Costa, J.-P. Gelas, Y. Georgiou, L. Lefevre, A.-C. Orgerie, J.-M. Pierson, O. Richard, and K. Sharma. The GREEN-NET Framework: Energy Efficiency in Large Scale Distributed Systems. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

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

[81] P. Dadam. *Verteilte Datenbanken und Client/Server-Systeme — Grundlagen, Konzepte und Realisierungsformen.* Springer Berlin/Heidelberg, 1996.

[82] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin. Automatic SQL Tuning in Oracle 10g. In *VDLB '04: Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 1098–1109. Morgan Kaufmann, 2004.

[83] N. H. Daudpota. Five Steps to Construct a Model of Data Allocation for Distributed Database Systems. *Journal of Intelligent Information Systems*, 11(2):153–168, September 1998.

[84] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[85] B. Devlin, J. Gray, B. Laing, and G. Spix. Scalability Terminology: Farms, Clones, Partitions, and Packs: RACS and RAPS. Technical Report MS-TR-99-85, Microsoft Research, 1999.

[86] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, 1992.

[87] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 228–237, 1986.

[88] S. Elhay, G. H. Golub, and J. Kautsky. Updating and Downdating of Orthogonal Polynomials with Data Fitting Applications. *SIAM Journal on Matrix Analysis and Applications*, 12(2):327–353, 1991.

[89] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 2nd Edition.* Benjamin/Cummings, 1994.

[90] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 117–130, 2006.

[91] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-Aware Load Balancing and Update Filtering in Replicated Databases. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 399–412, 2007.

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

[93] C. Faloutsos and P. Bhagwat. Declustering Using Fractals. In *PDIS '93: Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 18–25, 1993.

[94] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison Wesley, 1995.

[95] E. Fuchs. On Discrete Polynomial Least-Squares Approximation in Moving Time Windows. In *Proceedings of the Conference at the Mathematical Research Institute Oberwolfach*, volume 131 of *International Series of Numerical Mathematics*, pages 93–107, Basel, Switzerland, 1999. Birkhäuser.

[96] E. Fuchs, C. Gruber, T. Reitmaier, and B. Sick. Processing Short-Term and Long-Term Information With a Combination of Polynomial Approximation Techniques and Time-Delay Neural Networks. *IEEE Transactions on Neural Networks*, 2009. (accepted – to appear).

[97] E. Fuchs, T. Gruber, J. Nitschke, and B. Sick. On-line Motif Detection in Time Series With SwiftMotif. *Pattern Recognition*, 42(11):3015–3031, 2009.

[98] R. Gallersdörfer and M. Nicola. Improving Performance in Replicated Databases through Relaxed Coherency. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 445–456, 1995.

[99] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 444–455. VLDB Endowment, 2004.

[100] D. F. Garcia, J. Garcia, M. Garcia, I. Peteira, R. Garcia, and P. Valledor. Benchmarking of Web Services Platforms - An Evaluation with the TPC-App Benchmark. In *WEBIST '06: International Conference on Web Information Systems and Technologies*, 2006.

[101] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[102] S. L. Garfinkel. An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS. Technical Report TR-08-07, School for Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA, 2007.

[103] S. Ghandeharizadeh, D. J. DeWitt, and W. Qureshi. A Performance Analysis of Alternative Multi-Attribute Declustering Strategies. In *SIGMOD '92: Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 29–38, 1992.

[104] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[105] G. Graefe. Database Servers Tailored to Improve Energy Efficiency. In *SETMDM '08: Proceedings of the EDBT Workshop on Software Engineering for Tailor-Made Data Management*, pages 24–28, 2008.

[106] J. Gray. Notes on Data Base Operating Systems. In R. Bayer, R. Graham, and G. Seegmüller, editors, *Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer Berlin / Heidelberg, 1978.

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

[108] J. Gray. Database and Transaction Processing Performance Handbook. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann Publishers, 1993.

[109] J. Gray, P. Homan, H. F. Korth, and R. Obermarck. A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System. In *Berkeley '81: Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks*, page 125, 1981.

[110] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 243–252, New York, NY, USA, 1994. ACM.

[111] M. Gueye, I. Sarr, and S. Ndiaye. Database Replication in Large Scale Systems: Optimizing the Number of Replicas. In *EDBT/ICDT '09: Proceedings of the 2009 EDBT/ICDT Workshops*, pages 3–9, 2009.

[112] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[113] R. B. Hagmann and D. Ferrari. Performance Analysis of Several Back-End Database Architectures. *ACM Transactions on Database Systems*, 11(1):1–26, 1986.

[114] K. Haller. Towards the Industrialization of Data Migration: Concepts and Patterns for Standard Software Implementation Projects. In *CAiSE '09: Proceedings of the 21st International Conference on Advanced Information Systems Engineering*, pages 63–78, 2009.

[115] M. Hammer and B. Niamir. A Heuristic Approach to Attribute Partitioning. In *SIGMOD '79: Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 93–101, 1979.

[116] T. Härder. DBMS Architecture - the Layer Model and its Evolution. *Datenbank-Spektrum*, 13:45–57, 2005.

[117] T. Härder, V. Hudlet, Y. Ou, and D. Schall. Energy Efficiency Is Not Enough, Energy Proportionality Is Needed! In *FlashDB '11: The First International Workshop on Flash-based Database Systems*, pages 226–239, 2011.

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

[119] T. Heath, B. Diniz, E. V. Carrera, W. M. Jr., and R. Bianchini. Energy Conservation in Heterogeneous Server Clusters. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 186–195, New York, NY, USA, 2005. ACM.

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

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

[122] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey. Early Observations on the Performance of Windows Azure. In *HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 367–376, 2010.

[123] H. Hlavacs, K. A. Hummel, R. Weidlich, A. Houyou, A. Berl, and H. de Mee. Distributed Energy Efficiency in Future Home Environments. *Annals of Telecommunications*, 63(9-10):473–485, October 2008.

[124] J. E. Hoag and C. W. Thompson. A Parallel General-Purpose Synthetic Data Generator. *SIGMOD Record*, 36(1):19–24, 2007.

[125] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 35–43, 2005.

[126] J. A. Hoffer and D. G. Severance. The Use of Cluster Analysis in Physical Data Base Design. In *VLDB '75: Proceedings of the 1st International Conference on Very Large Data Bases*, pages 69–86, 1975.

[127] M. Holze and N. Ritter. Autonomic Databases: Detection of Workload Shifts with n-Gram-Models. In *ADBIS '08: Proceedings of the 12th East European conference on Advances in Databases and Information Systems*, volume 5207/2008 of *Lecture Notes in Computer Science*, pages 127–142, Berlin / Heidelberg, Germany, 2008. Springer-Verlag.

[128] T. Horvath, K. Skadron, and T. F. Abdelzaher. Enhancing Energy Efficiency in Multi-tier Web Server Clusters via Prioritization. In *IPDPS '07: 21th International Parallel and Distributed Processing Symposium*, pages 1–6. IEEE Computer Society, 2007.

[129] K. Houkjær, K. Torp, and R. Wind. Simple and Realistic Data Generation. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1243–1246. VLDB Endowment, 2006.

[130] W. W. Hsu, A. J. Smith, and H. C. Young. Characteristics of Production Database Workloads and the TPC Benchmarks. *IBM Systems Journal*, 40(3):781–802, 2001.

[131] V. Hudlet and D. Schall. Measuring Energy Consumption of a Database Cluster. In *BTW '11: Datenbanksysteme für Business, Technologie und Web, 14. Fachtagung des GI-Fachbereichs DBIS*, pages 734–737, 2011.

[132] T. Härder and A. Reuter. Concepts for Implementing a Centralized Database Management System. In *Proceedings of the International Computing Symposium*, 1983.

[133] T. Härder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[134] J. M. Kaplan, W. Forrest, and N. Kindler. Revoluzionizing Datacenter Efficiency. Technical report, McKinsey & Company, 2008.

[135] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *VLDB '00: Proceedings of 26th International Conference on Very Large Data Bases*, pages 134–143, 2000.

[136] A. Kemper and A. Eickler. *Datenbanksysteme - Eine Einführung.* Oldenbourg Verlag, 2009.

[137] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36:41–50, 2003.

[138] M. F. Khan, R. Paul, I. Ahmed, and A. Ghafoor. Intensive Data Management in Parallel Systems: A Survey. *Distributed and Parallel Databases*, 7(4):383–414, 1999.

[139] Y.-J. Kim, K.-T. Kwon, and J. Kim. Energy-Efficient File Placement Techniques for Heterogeneous Mobile Storage Systems. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, 2006.

[140] G. King. Running IBM System z at High CPU Utilization. Technical report, IBM Corporation, 2007.

[141] R. D. King, C. Feng, and A. Sutherland. StatLog: Comparision of Classification Algorithms on Large Real-World Problems. *Applied Artificial Intelligence: An International Journal*, 9(3):289–333, 1995.

[142] M. Koyutürk and C. Aykanat. Iterative-Improvement-Based Declustering Heuristics for Multi-Disk Databases. *Information Systems*, 30(1):47–70, 2005.

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

[144] E. Laczynski. Scaling Brands in the Cloud - Leveraging the Cloud for High-Traffic, High-Profile Web Marketing Events. Technical report, LTech, 2010.

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

[146] J. Leverich and C. Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. In *HotPower '09: Workshop on Power Aware Computing and Systems*, New York, NY, USA, 2009. ACM.

[147] P. J. Lin, B. Samadi, A. Cipolone, D. R. Jeske, S. Cox, C. Rendón, D. Holt, and R. Xiao. Development of a Synthetic Data Set Generator for Building and Testing Information Discovery Systems. In *ITNG '06: Proceedings of the Third International Conference on Information Technology: New Generations*, pages 707–712, Washington, DC, USA, 2006. IEEE Computer Society.

[148] J. W. Lloyd. Practical Advantages of Declarative Programming. In *GULP-PRODE '94: Proceedings of the Joint Conference on Declarative Programming*, 1994.

[149] M. Luhring, K.-U. Sattler, K. Schmidt, and E. Schallehn. Autonomous Management of Soft Indexes. In *ICDEW '07: Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 450–458. IEEE Computer Society, 2007.

[150] G. Marsaglia. Normal (Gaussian) Random Variables for Supercomputers. *The Journal of Supercomputing*, 5(1):49–55, 1991.

[151] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840, 2004.

[152] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority Mechanisms for OLTP and Transactional Web Applications. In *ICDE '04: Priority Mechanisms for OLTP and Transactional Web Applications*, pages 535–546, 2004.

[153] M. Mehta and D. J. DeWitt. Data Placement in Shared-Nothing Parallel Database Systems. *The VLDB Journal*, 6(1):53–72, 1997.

[154] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. Scale-up x Scale-out: A Case Study using Nutch/Lucene. In *IPDPS '07: International Parallel and Distributed Processing Symposium*. IEEE, 2007.

[155] J. M. Milan-Franco, R. Jiménez-Peris, M. P. no Martínez, and B. Kemme. Adaptive Middleware for Data Replication. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 175–194, 2004.

[156] S. Mintz and B. Cohen. Comparative Management Cost Study: Oracle Database 10g Release 2 and IBM DB2 Universal Database 9.1. Technical report, Edison Group, 2006.

[157] M. Mitzenmacher. A Brief History of Generative Models for Power Law and Lognormal Distributions. *Internet Mathematics*, 1(2):226–251, 2004.

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

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

[160] J. Muthuraj, S. Chakravarthy, R. Varadarajan, and S. B. Navathe. A Formal Approach to the Vertical Partitioning Problem in Distributed Database Design. In

*PDIS '93: Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 26–34, Washington, DC, USA, 1993. IEEE Computer Society.

[161] R. O. Nambiar and M. Poess. The Making of TPC-DS. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1049–1058, 2006.

[162] S. B. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical Partitioning Algorithms for Database Design. *ACM Transactions on Database Systems*, 9(4):680–710, 1984.

[163] S. B. Navathe, K. Karlapalem, and M. Ra. A Mixed Fragmentation Methodology For Initial Distributed Database Design. *Journal of Computer and Software Engineering*, 1995.

[164] J. Nowitzky. Partitionierungstechniken in Datenbanksystemen – Motivation und Überblick. *Informatik-Spektrum*, 24(6):345–356, 2001.

[165] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC '09: First TPC Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252, 2009.

[166] P. E. O'Neil. The Set Query Benchmark. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann Publishers, 1993.

[167] Oracle. *MySQL Cluster*, 2010. Extract from the MySQL 5.1 Reference Manual.

[168] O. Ozmen, K. Salem, J. Schindler, and S. Daniel. Workload-Aware Storage Layout for Database Systems. In *SIGMOD '10: Proceedings of the International Conference on Management of Data*, pages 939–950, 2010.

[169] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[170] S. Papadomanolakis and A. Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM '04: Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, pages 383–392, Washington, DC, USA, 2004. IEEE Computer Society.

[171] S. Papadomanolakis and A. Ailamaki. An Integer Linear Programming Approach to Database Design. In *ICDEW '07: Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshops*, pages 442–449, Washington, DC, USA, 2007. IEEE Computer Society.

[172] S. Papadomanolakis, D. Dash, and A. Ailamaki. Efficient Use of the Query Optimizer for Automated Physical Design. In *VLDB '07: Proceedings of the 33rd*

*international conference on Very large data bases*, pages 1093–1104. VLDB Endowment, 2007.

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

[174] D. A. Patterson, G. A. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988*, volume 17 of *SIGMOD Record*, pages 109–116. ACM, ACM Press, September 1988.

[175] C. Pedraza, E. Castillo, J. Castillo, C. Camarero, J. L. Bosque, J. I. Martínez, and R. Menendez. SMILE: Scientific Parallel Multiprocessing based on Low-Cost Reconfigurable Hardware. In *FPL '08: Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 277–278, 2008.

[176] V. Petrucci, O. Loques, and D. Mossé. A Framework for Dynamic Adaptation of Power-Aware Server Clusters. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1034–1039, New York, NY, USA, 2009. ACM.

[177] C. Plattner and G. Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 155–174, 2004.

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

[179] M. Poess. Controlled SQL Query Evolution for Decision Support Benchmarks. In *WSOP '07: Proceedings of the 6th International Workshop on Software and Performance*, pages 38–41. ACM, 2007.

[180] M. Poess and C. Floyd. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Record*, 29(2000):64–71, 2000.

[181] M. Poess, R. O. Nambiar, K. Vaid, J. John M. Stephens, K. Huppler, and E. Haines. Energy Benchmarks: A Detailed Analysis. In *e-Energy '10: Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, pages 131–140, 2010.

[182] M. Poess, R. O. Nambiar, and D. Walrath. Why You Should Run TPC-DS: A Workload Analysis. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1138–1149. VLDB Endowment, 2007.

[183] M. Pöss and C. Floyd. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Record*, 29(4):64–71, 2000.

[184] W. H. Press, S. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition edition, 2007.

[185] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.

[186] T. Rabl, C. Koch, , G. Hölbling, and H. Kosch. Design and Implementation of the Fast Send Protocol. *Journal of Digital Information Management*, 7(2):120–127, 2009.

[187] T. Rabl, A. Lang, T. Hackl, B. Sick, and H. Kosch. Generating Shifting Workloads to Benchmark Adaptability in Relational Database Systems. In R. O. Nambiar and M. Poess, editors, *TPCTC '09: First TPC Technology Conference on Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 116–131. Springer, 2009.

[188] T. Rabl, M. Pfeffer, and H. Kosch. Dynamic Allocation in a Self-Scaling Cluster Database. *Concurrency and Computation: Practice and Experience*, 20(17):2025–2038, 2007.

[189] E. Rahm. *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung*. Addison Wesley, 1994.

[190] R. Ramamurthy, D. J. DeWitt, and Q. Su. A Case for Fractured Mirrors. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 430–441. VLDB Endowment, 2002.

[191] P. Rubel, M. Gillen, J. Loyall, R. Schantz, A. Gokhale, J. Balasubramanian, A. Paulos, and P. Narasimhan. Fault Tolerant Approaches for Distributed Real-time and Embedded Systems. In *MILCOM '07: Military Communications Conference*, pages 1–8. IEEE, 2007.

[192] C. Rusu, A. Ferreira, C. Scordino, and A. Watson. Energy-Efficient Real-Time Heterogeneous Server Clusters. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 418–428, Washington, DC, USA, 2006. IEEE Computer Society.

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

[194] K. Sankaralingam and R. H. Arpaci-Dusseau. Get the Parallelism out of My Cloud. In *HotPar '10: Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, 2010.

[195] K.-U. Sattler, I. Geist, and E. Schallehn. QUIET: Continuous Query-Driven Index Tuning. In *VLDB '03 Proceedings of the 29th International Conference on Very Large Data Bases*, pages 1129–1132, 2003.

[196] D. Schall and V. Hudlet. WattDB: An Energy-Proportional Cluster of Wimpy Nodes. In *SIGMOD '11: Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2011.

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

[198] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: Continuous On-Line Tuning. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 793–795, 2006.

[199] K. Schnaitter, N. Polyzotis, and L. Getoor. Index Interactions in Physical Design Tuning: Modeling, Analysis, and Applications. *PVLDB*, 2(1):1234–1245, 2009.

[200] A. K. Seewald, J. Petrak, and G. Widmer. Hybrid Decision Tree Learners with Alternative Leaf Classifiers: An Empirical Study. In *FLAIRS '01: Proceedings of the Fourteenth International Florida Artificial Intelligence Research Society Conference*, pages 407–411, 2001.

[201] M. E. Senko. Data Structures and Data Accessing in Data Base Systems Past, Present, Future. *IBM Systems Journal*, 16(3):208–257, 1977.

[202] D. Serrano, M. P. no Martínez, R. Jiménez-Peris, and B. Kemme. Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, pages 290–297, 2007.

[203] S. Shankland. Google Spotlights Data Center Inner Workings. `http://news.cnet.com/8301-10784_3-9955184-7.html` (last visited 02.03.2011), May 2008.

[204] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, O. Fox, and D. Patterson. Cloudstone: Multi-Platform, Multi-Language Benchmark and Measurement Tools for Web 2.0. In *CCA '08: Proceedings of the 1st Workshop on Cloud Computing and its Applications*, 2008.

[205] G. Soundararajan, C. Amza, and A. Goel. Database Replication Policies for Dynamic Content Applications. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 89–102, 2006.

[206] J. M. Stephens and M. Poess. MUDD: a multi-dimensional data generator. In *WOSP '04: Proceedings of the 4th international workshop on Software and performance*, pages 104–109, New York, NY, USA, 2004. ACM.

[207] T. Stöhr, H. Märtens, and E. Rahm. Multi-Dimensional Database Allocation for Parallel Data Warehouses. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Databases*, pages 273–284, 2000.

[208] M. Stonebraker. The Case for Shared Nothing. *IEEE Database Bulletin*, 9(1):4–9, 1986.

[209] M. Stonebraker. A new Direction for TPC? In R. O. Nambiar and M. Poess, editors, *TPCTC '09: First TPC Technology Conference on Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 11–17. Springer, 2009.

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

[211] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 553–564. VLDB Endowment, 2005.

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

[213] M. Stonebraker and R. Cattell. Ten Rules for Scalable Performance in Simple Operation Datastores. *Communications of the ACM*, 2011. to appear.

[214] A. Syropoulos. Mathematics of Multisets. In *WMP '00: Proceedings of the Workshop on Multiset Processing*, volume 2235 of *Lecture Notes in Computer Science*, pages 347–358, Berlin, Germany, 2001. Springer.

[215] A. S. Szalay, J. Gray, A. Thakar, P. Z. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. vandenBerg. The SDSS SkyServer: Public Access to the Sloan Digital Sky Server Data. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 570–581, New York, NY, USA, 2002. ACM.

[216] A. Thomson and D. J. Abadi. The Case for Determinism in Database Systems. In *VLDB '10: Proceedings of the 36th International Conference on Very Large Data Bases*, pages 70–80. VLDB Endowment, 2010.

[217] A. Thusoo, J. S. Sarma, N. Jai, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive – A Petabyte Scale Data Warehouse Using Hadoop. In *ICDE '10: 26th IEEE International Conference on Data Engineering*, pages 996–1005, 2010.

[218] D. Tsichritzis and A. C. Klug. The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Dabatase Management Systems. *Information Systems*, 3(3):173–191, 1978.

[219] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the Energy Efficiency of a Database Server. In *SIGMOD '10: Proceedings of the International Conference on Management of Data*, 2010.

[220] J. van den Bercken and B. Seeger. An Evaluation of Generic Bulk Loading Techniques. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Databases*, pages 461–470, 2001.

[221] V. Venkatachalam and M. Franz. Power Reduction Techniques For Microprocessor Systems. *ACM Computing Surveys*, 37(3):195–237, 2005.

[222] S. Voß. Meta-heuristics: The State of the Art. In *ECAI '00: Proceedings of the Workshop on Local Search for Planning and Scheduling-Revised Papers*, volume 2148, pages 1–23. Springer Berlin / Heidelberg, 2001.

[223] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In *VLDB '02: Proceedings of the 28th International Conference on Very Large Data Bases*, pages 20–31. VLDB Endowment, 2002.

[224] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2001.

[225] C. B. Weinstock and J. B. Goodenough. On System Scalability. Technical Report CMU/SEI-2006-TN-012, Carnegie Mellon University, 2006.

[226] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, 1994.

[227] D. Wiese, G. Rabinovitch, M. Reichert, and S. Arenswald. Autonomic Tuning Expert: a Framework for Best-Practice Oriented Autonomic Database Tuning. In *CASCON '08: Proceedings of the 2008 conference of the center for advanced studies on collaborative research*, pages 27–41, New York, NY, USA, 2008. ACM.

[228] L. Wyatt, B. Caufield, and D. Pol. Principles for an ETL Benchmark. In *TPC TC '09: First TPC Technology Conference on Performance Evaluation and Benchmarking*, pages 183–198, 2009.

[229] S. B. Yao, S. B. Navathe, and J.-L. Weldon. An Integrated Approach to Dabatase Design. In S. B. Yao, S. B. Navathe, J.-L. Weldon, and T. Kunii, editors, *ata Base Design Techniques I*, volume 132 of *Lecture Notes in Computer Science*, pages 1–30. Springer Berlin / Heidelberg, 1982.

[230] E. Young, P. Cao, and M. Nikolaiev. First TPC-Energy Benchmark: Lessons Learned in Practice. In *TPCTC '10: Second TPC Technology Conference on Performance Evaluation and Benchmarking*, volume 6417 of *LNCS*, pages 136—-152, 2010.

[231] S. Zhou and M. H. Williams. Data Placement in Parallel Database Systems. In *Parallel Database Techniques*, pages 203–218. IEEE Computer Society, 1998.

[232] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB '04: Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 1087–1097. Morgan Kaufmann, 2004.

[233] M. E. Zorrilla, E. Mora, P. Corcuera, and J. Fernández. Vertical Partitioning Algorithms in Distributed Databases. In *EUROCAST '99: Computer Aided Systems Theory*, volume 1798 of *Lecture Notes in Computer Science*, pages 465–474. Springer, 1999.