Technische Universität München
Fakultät für Informatik
Lehrstuhl III – Datenbanksysteme

Community-Driven Data Grids

Diplom-Informatiker Univ.
Tobias Scholl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Hans Michael Gerndt

Prüfer der Dissertation:
1. Univ.-Prof. Alfons Kemper, Ph. D.
2. Univ.-Prof. Dr. Dieter Kranzlmüller,
   Ludwig-Maximilians-Universität München

Die Dissertation wurde am 17.09.2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 01.03.2010 angenommen.

To my daughter Julia Sophie

Abstract

E-science communities and especially the astronomy community have put tremendous efforts into providing global access to their distributed scientific data sets to foster vivid data and knowledge sharing within their scientific federations. Beyond already existing huge data volumes, the collaborative researchers face major challenges in managing the anticipated data deluge of forthcoming projects with expected data rates of several terabytes a day, such as the Panoramic Survey Telescope and Rapid Response System (Pan-STARRS), the Large Synoptic Survey Telescope (LSST), or the Low Frequency Array (LOFAR).

In this thesis, we describe and investigate *community-driven data grids* as an e-science data management solution. Community-driven data grids target at domain-specific federations and provide a scalable, distributed, and collaborative data management. Our infrastructure optimizes the overall query throughput by employing dominant data characteristics (e. g., data skew) and query patterns. By combining well-established techniques for data partitioning and replication with Peer-to-Peer (P2P) technologies, we can address several challenging problems: data load balancing, efficient data dissemination and query processing, handling of query hot spots, and the adaption to short-term query bursts as well as long-term load redistributions.

We propose a framework for investigating application-specific index structures to create locality-aware partitioning schemes (so-called histograms) and to find appropriate data mapping strategies. We particularly investigate how far mapping strategies based on space filling curves preserve query locality and achieve data load balancing depending on query patterns in comparison to a random mapping.

An efficient data dissemination technique for the anticipated large data volumes is important for several use cases within scientific federations, including initial data distribution and data replication. A scalable solution should neither induce a high load on the transmitting servers nor create a high messaging overhead. Optimizing data distribution with regards to latency and bandwidth is infeasible in our scenario. Therefore, we propose several strategies that optimize network traffic, use chunk-based feeding, and improve data processing at receiving nodes in order to speed up data feeding.

In the face of different typical submission scenarios, we show how community-driven data grids can adapt their query coordination strategies during query processing. We explore the impact of uniform of skewed submission patterns and compare multiple strategies with regards to their usability and scalability for data-intensive applications. Our techniques improve query throughput considerably by increased parallelism and data load balancing in both local as well as wide area deployments.

Addressing skewed query workloads, so-called query hot spots, by query load balancing and directly meet the requirements of a data-intensive e-science environment is another interesting and challenging task. We enhance our data-driven partitioning schemes to trade off data load balancing against handling query hot spots via splitting and replication. We use a cost-based approach for workload-aware data partitioning. Based on these workload-aware partitioning schemes, we use master-slave replication to compensate for short-term peaks in query load and address long-term shifts in data and query distributions by partitioning scheme evolution.

Our research prototype HiSbase realizes the concepts described within this thesis and offers a basis for further research shaping the data management of future scientific communities.

Acknowledgements

First of all, I am grateful to my advisor Prof. Alfons Kemper, Ph. D., for giving me the opportunity to pursue this thesis under his guidance. During many discussions, he provided invaluable advice, comments, and encouragements. I also thank Prof. Dr. Dieter Kranzlmüller from the Ludwig-Maximilians-Universität München for serving as reviewer for my thesis.

During my time at the database group at TUM, I enjoyed working with my colleagues, especially Dr. Angelika Reiser who coordinated our efforts in the AstoGrid-D project and had an inexhaustible supply of knowledge and experience. For their help, the pleasant working atmosphere, and insightful discussions, I thank Martina-Cezara Albutiu, Stefan Aulbach, Veneta Dobreva, Dr. Daniel Gmach, Prof. Dr. Torsten Grust, Benjamin Gufler, Sebastian Hagen, Dean Jacobs, Ph. D., Stefan Krompaß, Dr. Richard Kuntschke, Manuel Mayr, Jessica Müller, Fabian Prasser, Jan Rittinger, Andreas Scholz, Michael Seibold, Dr. Bernhard Stegmaier, Dr. Jens Teubner, and Dr. Martin Wimmer. I particularly thank Evi Kollmann, our secretary.

Several students offered their support and devotion to implement our research prototype HiSbase. I thank Daniel Weber for supporting the development of the first prototype. Bernhard Bauer helped implementing and evaluating the quadtree-based histograms and the workload-aware partitioning schemes. Achim Landschoof implemented parts of our framework for comparing histograms and Dong Li implemented a statistics component to measure network traffic. Ella Qiu implemented the query coordinator selection strategies during her RISE internship, which was sponsored by the DAAD and TUM. Tobias Mühlbauer was a great support during the implementation and evaluation of the data feeding strategies. I also thank my colleagues Benjamin Gufler and Jessica Müller for their contributions to the HiSbase project.

The HiSbase project is part of the AstroGrid-D project and is funded by the German Federal Ministry of Education and Research (BMBF) within the D-Grid initiative under contract 01AK804F. I thank Dr. Thomas Fuhrmann for providing access to the PlanetLab test bed and the LRZ Grid team for their great support and resources.

Finally, I thank my wife Nina and my parents Elisabeth and Hartmut for their love, support, and endurance throughout the years.

Munich, September 2009 Tobias Scholl

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

E-science projects of many research communities, e. g., biology, the geosciences, high-energy physics, or astrophysics, face huge data volumes from existing experiments. Due to the expected data rates of upcoming astrophysical projects, e. g., the Panoramic Survey Telescope and Rapid Response System (Pan-STARRS)[1], producing several terabytes a day, current centralized data management approaches offer only limited scalability.

Combining and correlating information from various experiments or observations are the key to finding new scientific insights. Mostly, the institutions conducting the experiments provide the data results to the whole community by hosting the data on their own servers. This approach of autonomous data management is not well-suited for the application scenario just described as each data source needs to be queried individually and (probably large) intermediate results need to be shipped across the network.

In astronomy, for example, most often the individual projects provide interfaces to their own data set for interactive or service-based data retrieval. These service interfaces are standardized by the International Virtual Observatory Alliance (IVOA)[2] in order to ensure interoperability between the various interfaces. User queries can consume only a limited amount of CPU resources, have a result size limit, and the number of parallel queries per user is restricted in order to allow fair use and to avoid overloading the servers. Examples of such restrictions are query cancellation after 10 minutes running time or a maximum result size of 100 000 rows. Batch systems (such as CasJobs (O'Mullane et al., 2005)) offer less restrictive access to the data sources and sometimes even a private database for later processing or sharing the results with colleagues. However, some queries might suffer from long queuing times.

Furthermore, we observe that in many e-science communities, data sets are highly skewed and scientific data analysis tasks exhibit a large degree of spatial locality. Dealing with *data skew* while *preserving spatial locality* is fundamental to realize a scalable information infrastructure for these communities. Section 1.2 gives a more detailed scenario from the astrophysics domain exhibiting these characteristics.

[1]http://pan-starrs.ifa.hawaii.edu

[2]http://www.ivoa.net

To avert the scalability issues of their current systems, communities investigate different technologies. The adaption to domain-specific data and query characteristics is fundamental for these approaches to result in benefits for the researchers. These characteristics can include properties such as data skew and complex multi-dimensional range queries.

Future e-science communities require the efficient processing of data volumes that centralized data processing or a data warehouse approach cannot sufficiently scale up to. Centralized data processing, where researchers ship data on demand from the distributed sources to a processing site—most often their own computer—has the deficiency of high transmission cost. On the other hand, a data warehouse does not cope with the high query load and the demanding throughput requirements.

1.1 Problem Statement

In order to deal with the sheer size of their resulting data, researchers within a community join forces in *Virtual Organizations* and build infrastructures for their scientific federations, so-called *data grids* (Venugopal et al., 2006). These data grids interconnect dedicated resources using high-bandwidth networks and enable researchers to share and correlate their data sets within the community. In order to ensure reproducibility, published data sets are not changed. Instead, new additional versions are made available. Moreover, an *increasing popularity* within the user community puts high demands on the various architectural design choices, such as providing high query throughput. Further challenging aspects are *skewed data distributions* in the data sets as well as *query hot spots*.

E-Science communities need support for building an infrastructure for data sharing that 1) is able to directly deal with several terabytes or even petabytes of data, 2) integrates the existing high-bandwidth networks with several hundred nodes within the communities, and 3) offers high throughput to cope with a steadily growing user community. Given these requirements, *how can we provide a scalable infrastructure that is capable of using the shared resources and performs data as well as query load balancing*?

1.2 Application Setting

In astrophysics as well as in other scientific communities we expect exponential data growth rates in addition to already existing enormous data volumes. Furthermore, the increasing access rates by researchers to these information systems support the need for a scalable and efficient data management. The correlation and combination of observational data or data gained from scientific simulations (e. g., covering different wave bands) is the key for gaining new scientific insights. The creation of likelihood maps for galaxy clusters (Carlson et al., 2007; Schücker et al., 2004) or the classification of spectral energy distributions (Kuntschke et al., 2006) are examples for such applications. Together with our cooperation partners from the AstroGrid-D community project (Enke et al., 2007) within the D-Grid initiative, we construct a grid environment that supports users in bringing their everyday science to the grid. Many typical astrophysical applications have been identified and in order to successfully port them to the grid, we developed a collection of grid tools and services. These applications range over distributed computation-intensive simulations or data-analysis tasks, steering robotic telescopes, and complex parallel workflows. Our main focus are data-intensive applications that access scientific databases from the grid or use grid-based data stream management (Kuntschke et al., 2006).

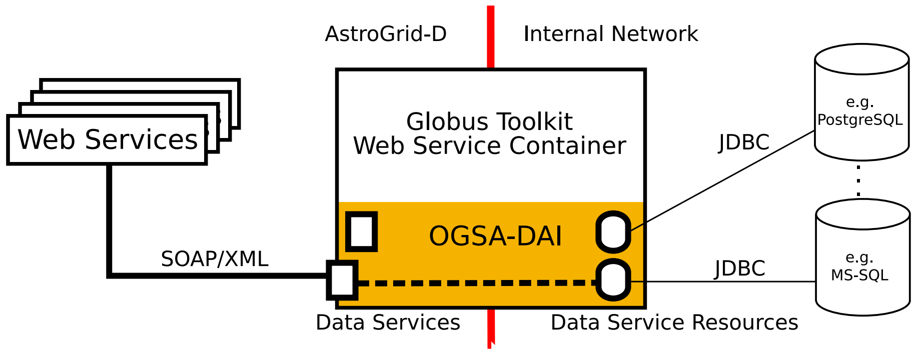


Figure 1.1: Database access within AstroGrid-D via the OGSA-DAI middleware

Catalog	Number of objects	Approx. object size	Size
SDSS (DR5)	215 million	14 KB	3.6 TB
TWOMASS	471 million	2 KB	1 TB
USNO-B1.0	1 000 million	0.9 KB	0.08 TB

Table 1.1: Size of current astronomical data sets

We therefore use data and workloads from the domain of astrophysics, though the techniques presented in this thesis are also applicable to other domains.

Within AstroGrid-D, we access persistent data, such as scientific databases, using the components developed by the *Open Grid Services Architecture – Data Access and Integration (OGSA-DAI)* project (Antonioletti et al., 2005). The OGSA-DAI project integrates with the Globus Toolkit and participates in the standardization process of the Open Grid Forum (OGF) *Data Access and Integration Services (DAIS)* working group (Antonioletti et al., 2006). OGSA-DAI offers a unified way of accessing and integrating distributed, heterogeneous data resources using web services or grid services. The flexible OGSA-DAI interface allows for integration of resources such as files, RDF graphs, and relational and XML databases in particular.

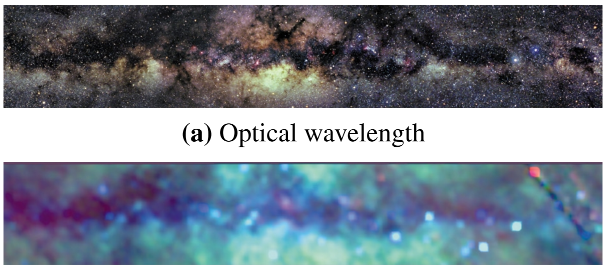
Figure 1.1 gives an overview of the main OGSA-DAI components and how they are used within AstroGrid-D. Integrated into a Globus Web Service Container, OGSA-DAI publishes a *Data Service* interface that allows web service or grid service clients to interact with *Data Service Resources*. These data service resources are databases exposed via the data service. OGSA-DAI can therefore hide the complexity of the individual database (e. g., driver, connection URL) from the grid user, the database resources can be kept behind institutional firewalls, and database access is secured using the mechanisms based on certificates provided by the Globus Toolkit. If communities want to integrate a new type of resource, they can define an interface for that resource via the OGSA-DAI activity-framework.

To give an idea of the future scalability challenges, Table 1.1 summarizes the number of objects, the approximate size of an individual object, and the complete size for three of the major current astrophysical data sets, the Sloan Digital Sky Survey (SDSS)[1], the Two Micron All Sky Survey (TWOMASS)[2], and the USNO-B1.0 [3] catalog. Assuming a uniform distribution of the catalogs in Table 1.1 to a thousand dedicated nodes of an astrophysics community grid, each node covers about 5 GB of data and thus the data sets fit almost completely in main memory.

[1]http://www.sdss.org/dr5/
[2]http://www.ipac.caltech.edu/2mass/
[3]http://www.nofs.navy.mil/data/fchpix/cfra.html

Project	Data growth rate	
	Per day	Per year
Pan-STARRS	10 TB	4 PB
LSST	18 TB	7 PB
LOFAR	33 TB	12 PB
LHC	42 TB	15 PB

Table 1.2: Estimated data grow rates for upcoming e-science projects



(a) Optical wavelength

(b) X-ray wavelength

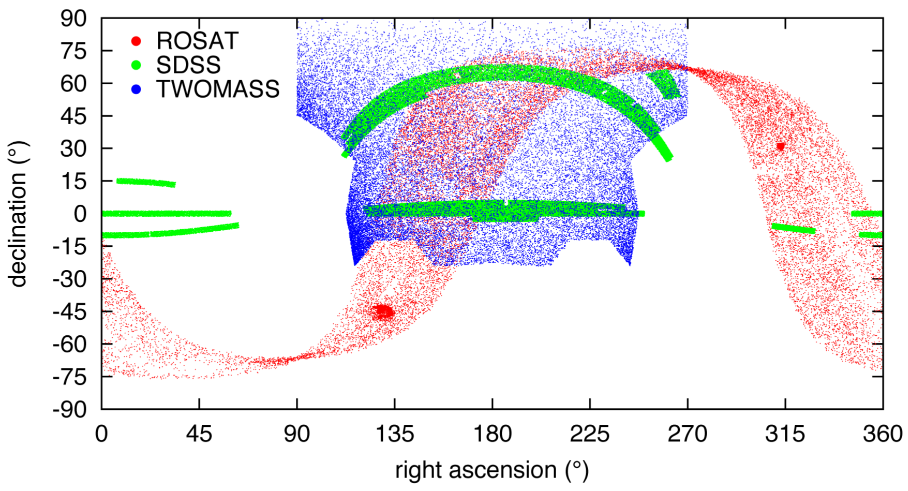
Figure 1.2: A multi-wavelength view on the milky way (source: http://mwmw.gsfc.nasa.gov/)

These data sets still could be managed at a single site, although with restrictions such as high transmission costs or limited resource availability. Upcoming e-science projects (see Table 1.2) in astrophysics and high energy physics face a data deluge which will be distributed across several sites. Examples for such upcoming projects beyond the Pan-STARRS project are the Large Synoptic Survey Telescope (LSST)[1] and the Low Frequency Array (LOFAR)[2] in astrophysics as well as the Large Hadron Collider (LHC)[3] in high energy physics.

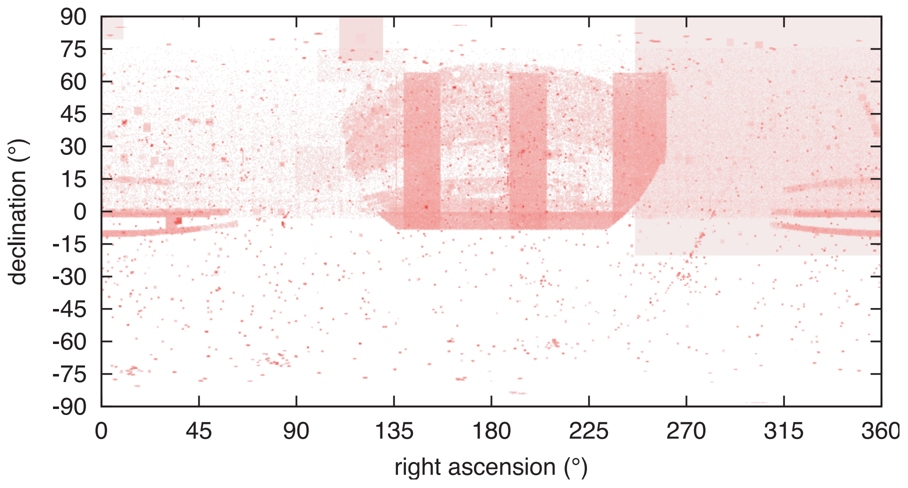
Researchers usually access and analyze logically related subsets of these data volumes. The restrictions of such subsets are mostly based on specific data characteristics. Typical access patterns over astrophysical data sets are point-near-point queries, point-in-region queries, and nearest-neighbor-searches. Such queries are usually *region-based*, i. e., they process data within certain regions of the sky. These regions are specified by the two-dimensional celestial coordinates, *right ascension (ra)* and *declination (dec)*. Region-based queries can, of course, also contain predicates on attributes other than the celestial coordinates. In case of celestial objects, other attributes might comprise detection time, catalog-identifier, temperature, or energy level. We use *cross-match* queries (Gray et al., 2006) as example for such region-based queries. Astrophysicists use cross-matching to determine whether data points from different archives are likely to stem from the same celestial object. Researchers take several point sources from an area (e. g., the milky way in Figure 1.2) in one data set and look for matching sources in other data sets.

Our astrophysics cooperation partners provided us with several data samples of some of the major observational catalogs in order to develop our system. The observational data set P_{obs} comprises about 137 million objects from subsets of the ROSAT (25 million objects), SDSS (84 million objects), and TWOMASS (28 million objects) catalogs and has a size of about 50 GB. Figure 1.3(a) shows the actual distribution of the three data samples, displaying the right ascension on the x-axis and the declination on the y-axis. The value range for right ascension

[1]http://lsst.org/
[2]http://www.lofar.org/
[3]http://lhc.web.cern.ch/lhc/



(a) The observational data set P_{obs} consisting of data samples from three catalogs showing data skew, i. e., a combination of densely populated areas wit data from all catalogs and areas without any data



(b) The observational query set Q_{obs} exhibiting query hot spots, i. e., several areas with many queries such as the intersection of the three catalogs

Figure 1.3: The observational data set and query set

and declination is $[0°, 360°[$ and $[-90°, 90°]$, respectively.

For P_{obs}, we constructed the corresponding observational query set Q_{obs} from real queries submitted to the web interface[1] of the SDSS catalog in August 2006.[2] We translated the original cone searches (with a circular search area) to queries having a square search area. For each query, we used the same midpoint and an edge length corresponding to the diameter of the circular search area. Queries using the default search parameters of the web interface make up 12% of the entire query log. Thus, we removed that particular query from our query set and used the remaining 1 100 000 queries during our evaluation. Figure 1.3(b) shows the queried areas and we clearly see that the workload is non-uniform and exhibits many query hot spots. Remarkably, these hot spots are in areas where the catalogs overlap.

The second data sample P_{mil} (Figure 1.4) originates from the Millennium simulation con-

[1] http://cas.sdss.org/astrodr7/en/tools/search/

[2] The query trace was kindly provided by Milena Ivanova and Martin Kersten from CWI, Amsterdam. It is the same workload used for their experience report on migrating SkyServer on MonetDB (Ivanova et al., 2007).

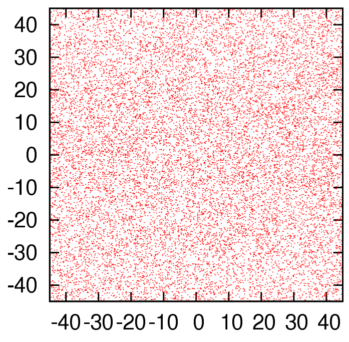


Figure 1.4: The uniform data set P_{mil} from the Millennium simulation

ducted by Springel et al. (2005) and consists of 160 million objects that are uniformly distributed on the area $[-45°, 45°] \times [-45°, 45°]$. This uniform data set allows us to create different workloads in order to investigate how histograms perform on uniform data and the impact of skewed query workloads without being influenced by data skew.

Throughout the discussion and in our prototype we use the relational data model and SQL for several reasons. First, large parts of the scientific data sets are stored within relational databases, though most data providers offer additional, e. g., form-based, interfaces for researchers not familiar with SQL. Moreover, the current specification for the IVOA *Astronomical Data Query Language (ADQL)* is based on the SQL standard SQL92. ADQL specifies additional functions that are fundamentally important for astrophysics research and require unification across several implementations.

1.3 Our Approach and Contributions

Community-driven data grids enable communities to individually address the two major issues, *high-throughput data management* and *correlation of distributed data sources*.

Scalable data sharing for e-Science grids

With HiSbase, our prototypical implementation of a community-driven data grid, we explore design alternatives for data grids between the both extremes of a centralized community warehouse and a fully-replicated data grid. We propose a decentralized and scalable approach to scientific data management using the existing available capacities—both CPU and main memory—of the community network resources. Based on distributed hash tables (DHTs), we partition data according to predominant query patterns and not according to the original data source. The symmetry of these networks, i. e., the fact that nodes act as servers (providing data) and as clients (issuing queries), offers increased fault-tolerance and robustness. In a DHT system, nodes automatically detect node failures and fix the overlay communication.

Preserving locality and handling data skew through domain specific partitioning

We suggest to reconsider static partitioning schemes as an application domain specific hash function to allow scalable information management in e-science communities. Occasionally, this hash function is updated to accommodate better load-balancing, just like database systems regularly update query optimizer statistics. In case of multi-dimensional data, space filling curves preserve the spatial locality between closely related data. HiSbase targets collaborative communities having vast data volumes with fairly stable data distributions. Long-term distribution changes can also be leveled by reorganizing the histogram.

Increased Query Throughput by parallelism and workload-awareness

We investigate the potential offered by P2P networks for increasing query throughput in data-intensive e-science applications. Achieving sufficient query throughput constitutes one of the main deficiencies of centralized data management. Moreover, by enhanced workload-awareness during the creation of the partitioning scheme and during runtime additional throughput improvements are possible.

1.4 Outline

In Chapter 2, we provide the reader with a general overview of HiSbase and describe the general design choices and decisions. Parts of this section have been presented at BTW 2007 (Scholl et al., 2007b), VLDB 2007 (Scholl et al., 2007a), and in an overview article in the FGCS journal (Scholl et al., 2009a).

The following chapters provide a more detailed discussion about the individual parts that orchestrate the data management infrastructure in community-driven data grids. The Chapters 3 through 6 describe the basic building blocks.

Chapter 3 focuses on the training phase where the partitioning scheme for distributing the data among the resources is built. This material has been presented at the IEEE e-Science Conference 2007 (Scholl et al., 2007c).

We describe how the data partitions are mapped to the nodes participating in a community-driven data grid in Chapter 4 and present our bulk feeding techniques for efficient data dissemination in Chapter 5.

Chapter 6 describes collaborative query processing and coordination and presents the setup and results of our throughput experiments within a community-driven data grid. The evaluation of the different coordination strategies has been presented at HPDC 2009 (Scholl et al., 2009c).

How the training phase can be extended to address also workload-awareness is described in Chapter 7. The results have been presented at EDBT 2009 (Scholl et al., 2009b).

We then describe our concepts for achieving further load balancing within community-driven data grids during runtime (Chapter 8). Parts of this chapter have been presented in the Datenbank-Spektrum (Scholl et al., 2008) and the Proceedings of the VLDB Endowment (Scholl and Kemper, 2008).

CHAPTER 2

HiSbase

This chapter gives an overview of the basic principles behind community-driven data grids. It provides enough background to selectively choose one of the following chapters to receive more detailed information about a particular aspect.

Our prototype HiSbase defines a distributed information infrastructure that allows the sharing of CPU resources and storage across scientific communities to build a community-driven data grid. We distribute data across nodes according to predominant query patterns to achieve high throughput for data analysis tasks. Therefore, most processing tasks can be performed locally, achieving high cache locality as nodes mainly process queries on logically related data hosted by themselves as illustrated in Figure 2.1. In the figure, the same geometric shapes denote logically related data originating from (possibly) different distributed sources. HiSbase partitions and allocates data fed into the system by means of community-specific distribution functions, called *histograms*. Thereby, related data objects of various sources are mapped to the same nodes. The original archive servers, which still serve as data sources, are complemented by our HiSbase infrastructure for high throughput query processing. In Section 2.1, we show a candidate data structure that preserves spatial locality and adapts to the data distribution.

HiSbase builds on the advances and maturity of distributed hash table (DHT) implementations (e. g., Stoica et al., 2001) which were devised in order to provide a scalable and failure-resilient data management for large-scale, distributed information systems. Objects and nodes are randomly mapped to a one-dimensional key space, e. g., by using a secure hash algorithm on either the IP address of a node or on a checksum of the content of an object. Nodes are responsible for a subspace of the key space and thus data load balancing is achieved. The initial versions of these protocols only offer scalable query access for exact-match queries.

HiSbase partitions multi-dimensional e-science data across an initial set of nodes for data load balancing. Later on, of course, additional resources (data and nodes) can be added to the community network, which is constructed as follows.

- We precompute the histogram of the actual data space in a preparatory *training phase* based on a training set and pass it to the initial HiSbase node during startup (Section 2.2.1).

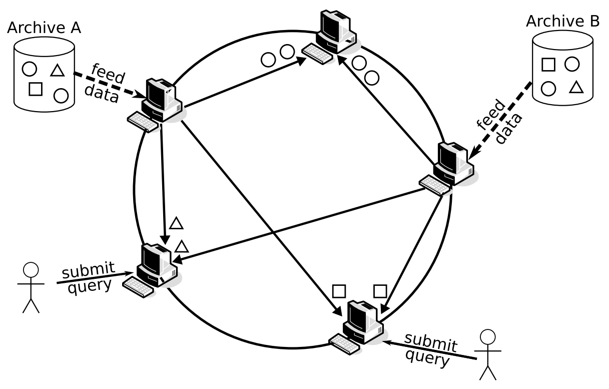


Figure 2.1: Architecture for community-driven data grids

- Additional nodes subsequently joining the network receive their own local copy of the histogram from a neighboring HiSbase node.
- HiSbase allocates data according to the precomputed histogram (Section 2.2.2) and uses the histogram as a routing index. Data archives feed data into HiSbase by sending their data to any HiSbase node which routes the data to the responsible node (Section 2.2.3).
- Every HiSbase node accepts queries and routes them to a coordinator node which may own (some of) the data needed to process the query. If the coordinator does not cover all the data relevant to the query, it guides cooperative query processing among all nodes contributing to the query result (Section 2.2.4).

In order to position HiSbase within the context of related scientific results from the literature, we conclude this chapter with a general discussion of related work (Section 2.3). We focus especially on works that are comparable in general and discuss specific related results within the individual chapters wherever appropriate.

2.1 Locality Preservation

To allow efficient query processing on logically related data sets we need to *preserve the locality of data*. Data locality is especially important for the performance of data analysis tasks in astrophysics. Distributing data objects randomly across a global information network severely impairs the performance of astrophysical query patterns.

2.1.1 Data Skew

Many application domains have highly skewed data sets. This skew originates from data spaces with a mix of densely and sparsely populated regions. The differences in data density may arise from the original data distribution or from the fact that some regions have been investigated more extensively than others, i. e., more data has been collected and is available. In astrophysics, celestial objects are not distributed uniformly over the sky, e. g., considering high data density in the galactic plane or a supernova. We use an abstract skewed data sample (Figure 2.2) for illustration.

In HiSbase, we preserve spatial proximity to efficiently process region-based queries (Section 2.2.1) while addressing the imbalance of the data distribution. HiSbase achieves this goal

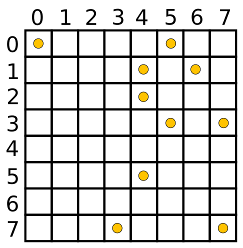


Figure 2.2: Sample data space with skewed data distribution

by calculating a histogram that equips the data grid with a community-specific data distribution. Among others, we describe the Z-quadtree histogram data structure that we designed to preserve spatial locality for astrophysics data sets. Z-quadtrees are *quadtrees* whose leaves correspond to histogram buckets and are linearized on the key space of the DHT using a space filling curve. These trees provide efficient access to histogram buckets (regions) while balancing the data load across data nodes.[1] Section 2.2.5 outlines how we additionally consider query load balancing within HiSbase.

2.1.2 Histogram Data Structures

HiSbase enables communities to design data structures for distributing their data across several nodes and to adapt to data and query characteristics of that particular community. We call these data structures *histograms* for their similarities to standard histograms. Histograms are, for example, commonly used in relational database management systems as means for selectivity estimations (Poosala et al., 1996).

Within HiSbase, a histogram is used in order to look up multi-dimensional areas and data points.

lookupArea(h,a) : S This function plays a central part during query processing. Given a multi-dimensional data area a, *lookupArea* returns the set S of region identifiers of the histogram h which intersect with a.

lookupPoint(h,p) : r Mainly used during data distribution, *lookupPoint* returns the region identifier r of the histogram h which contains a multi-dimensional data point p.

Most of the following histogram data structures are inspired by the intensive research conducted by the computer science community on locality-aware data structures developed for accessing and efficiently storing multi-dimensional data (Gaede and Günther, 1998; Samet, 2006). The individual community is free to choose any data structures implementing the interface required by HiSbase. Therefore, we are strengthening the histogram-related aspect rather than the aspect of indexing multi-dimensional data.

Z-quadtree: A Histogram Based on Quadtrees

The shape of data partitions defined by candidate data structures should be simple (e. g., squares). This allows simple (SQL) queries to retrieve data during the process of integrating new nodes (Section 2.2.2).

[1]In the following, we use the terms *regions* and *histogram buckets* interchangeably. The *leaves* of a Z-quadtree represent the histogram buckets for that particular histogram data structure.

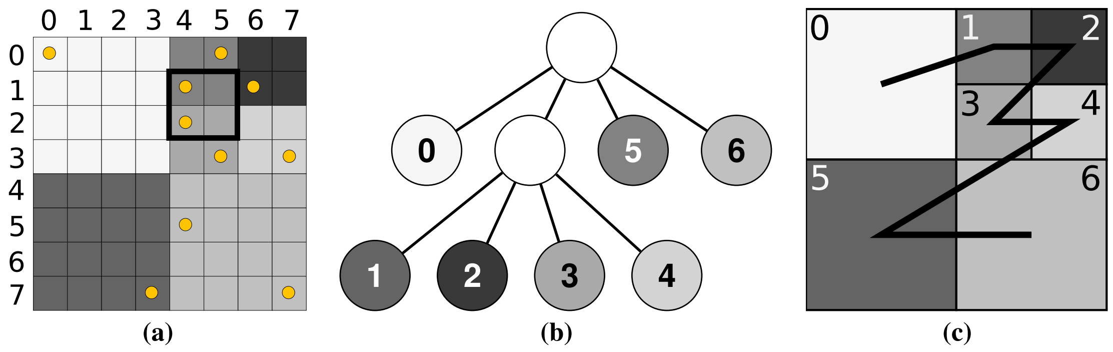


Figure 2.3: Application of the Z-quadtree to the data sample: (a) the Z-quadtree regions, (b) the corresponding quadtree, and (c) the linearization of its leaves

In the following, we describe the *Z-quadtree* as our preferred data structure, which is inspired by *quadtrees* (Samet, 1990).

A Z-quadtree partitions the data space according to the principle of recursive decomposition. For a d-dimensional data space, a Z-quadtree node is either a leaf with a d-dimensional data region or an inner node with 2^d children. The leaves of the quadtree correspond to the histogram buckets. After the Z-quadtree buckets are calculated they are linearized using the Z-order space filling curve (Orenstein and Merrett, 1984).

The *linearization* is then used to map the buckets on the key space of the underlying fabric. We use a space filling curve instead of a random mapping as the curve preserves spatial proximity if one node covers several buckets. If buckets are adjacent, they are likely to be managed by the same node.

Starting with a single leaf covering the entire data space, we sequentially insert the training set into the tree (Section 2.2.1). If the number of objects in the area of a leaf exceeds a predefined threshold, representing its capacity, the leaf is split into 2^d subareas according to the quadtree splitting strategy. Inner nodes forward the objects to the corresponding child. In Figure 2.3(a), we show the decomposition of our two-dimensional example data set of Figure 2.2 using a leaf capacity of two objects. After the complete training set is inserted, each leaf is assigned a *region identifier* using a depth-first search (Figure 2.3(b)). This immediately gives the desired leaf linearization which is shown in Figure 2.3(c). While using the Z-order is the canonical leaf linearization, other space filling curves such as the Hilbert curve (Hilbert, 1891) are also applicable.

Algorithm 2.1 describes how the set S of region identifiers that intersect with a query area a is retrieved in a Z-quadtree h. Starting at the root node, *lookupArea* is executed recursively. If the region r_n of a leaf n intersects with query area a, its region identifier $r_n.id$ is added to the result set S. Intersecting inner nodes invoke *lookupArea* on every subtree. The method to find the region which contains a data point, *lookupPoint*, can be realized similarly.

Z-quadtrees use the same concept as *linear quadtrees* (Gargantini, 1982), a data structure used in image encoding. Using a lower resolution for sparsely populated data subspaces in Z-quadtrees corresponds to compressing the representation for common subpixels of the linear quadtrees.

In contrast to the original quadtree, which is a spatial index structure, the Z-quadtree is used for data dissemination, as a routing index, and during query processing. The actual training data used to create a histogram is not stored in the data structure distributed to all nodes.

We present a detailed discussion of several histogram data structures in Chapter 3.

Algorithm 2.1: Z-quadtree implementation of $lookupArea(h,a)$

```
**Input:** Z-quadtree $h$ with root node $n_{root}$, query area $a$
**Output:** Set $S = \{\text{region id } r.id \mid \text{region } r \text{ intersects with } a\}$
  $S \leftarrow \{\}$
  $n \leftarrow n_{root}$
  if region $r_n$ of $n$ intersects with $a$ then
    if $n$ is leaf then
      $S \leftarrow S \cup \{r_n.id\}$
    else /* $n$ is inner node */
      for all subtrees $h_{child}$ of $n$ do
        $S \leftarrow S \cup lookupArea(h_{child}, a)$
      end for
    end if
  end if
```

Algorithm 2.2: Publish data in HiSbase

```
**Input:** Histogram $h$, multi-dimensional data point $p$
  Region id $r \leftarrow lookupPoint(h,p)$
  Send $newPointMessage(p)$ to $r$.
```

2.2 Architectural Design

The architectural design of HiSbase offers scientific researchers a framework for data and resource sharing within their community. Algorithms 2.2 and 2.3 formally define the interface for data publication and access within HiSbase.

In this section, we outline the creation of histograms during the training phase and the information maintained at HiSbase nodes. Finally, we describe data publication and node collaboration during query processing.

2.2.1 Training Phase (Histogram Build-Up)

Extracting the training samples, defining the partitioning of the data space, and distributing the partitions to the data nodes comprise the three steps of our training phase.

For constructing the histogram, data from each data source is taken into account. We can either use the entire data archive or a representative subsample. However, transmitting the entire data archive for histogram extraction is presumably prohibitive. For example, the subset could be extracted using a random sample. During our experiments, we achieved good histograms using 10% data samples in our *a priori* analysis. The data distribution does not change significantly very often (e. g., once a year), which makes such an *a priori* analysis applicable.

After the training set is inserted into the histogram, we serialize the histogram structure for distribution within the network. The training data itself is discarded.

The resulting histogram is passed to the initial node in the HiSbase network. Nodes subsequently joining the network receive the histogram from any other node in the network, mostly from one of their neighboring nodes. Hence, each node keeps a copy of the histogram.

The number of histogram regions is determined before the training phase. In our experiments, we used histograms with up to 250 000 regions in order to enable a flexible distribution

Algorithm 2.3: Query data in HiSbase

```
Input: Histogram $h$, multi-dimensional query area $a$.
    Set $S_R$ of relevant region ids $\leftarrow$ $lookupArea(h,a)$
    Select $coordinator$ $r_c$ from $S_R$
    Send $newQueryMessage(a,S_R)$ to $r_c$.
```

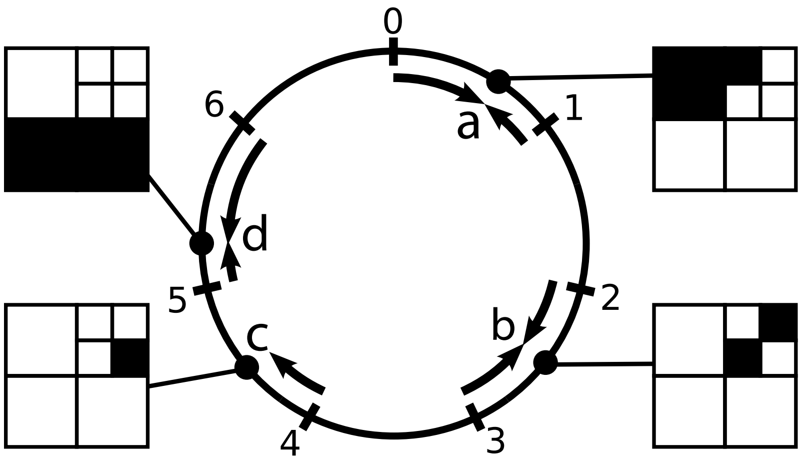


Figure 2.4: Mapping of the quadtree of Figure 2.3 to multiple nodes

of the individual data regions. The memory requirements of a histogram are small compared to the amount of data transmitted during query processing. As nodes presumably get their histogram from a physical neighbor, histogram distribution adds little overhead to the setup phase of the HiSbase network.

2.2.2 HiSbase Network

While the overall design of HiSbase abstracts from the underlying DHT implementation, we use the DHT infrastructure *Pastry* (Rowstron and Druschel, 2001) to manage nodes and route messages in HiSbase. Like Chord (Stoica et al., 2001), Pastry maps data and nodes to a one-dimensional key ring. In contrast to Chord, Pastry optimizes the initial phase of routing by preferring physical neighbors to speed up communication within the overlay network.

Mapping Nodes to Regions

The histogram regions are uniformly mapped onto the DHT ring identifiers. Due to this uniform distribution, all regions are mapped to a node with equal probability regardless of their individual size. The size of regions might vary due to the adaption to data skew. The nodes get a random identifier and are responsible for regions close to their identifier. Pastry, for example, uses 160-bit identifiers and ordinary comparisons in order to determine the closeness of identifiers. Figure 2.4 illustrates the evenly distributed regions (0–6) and their mapping to randomly distributed nodes (*a*, *b*, *c*, *d*) on the DHT key space. We use the routing of the underlying DHT system to automatically assign regions to nodes. To ensure that messages destined for a specific region are received by the appropriate node, we use the region identifiers for message routing.

We prefer to use the key-based routing functionality of the underlying DHT infrastructure over using a direct mapping of histogram buckets on nodes or using a centralized directory for

Algorithm 2.4: Handling node arrivals

Node p covers a set P of regions. Let P_{new} be the set of regions which node p is responsible for after a new node has arrived. The area a_i is the area of region i.

```
if $P_{new} \neq P$ then
  find $P_{move} = P \setminus P_{new}$
  for all $r \in P_{move}$ do
    $a_r = getArea(r)$
    redistribute data from $a_r$ to region $r$
  end for
end if
```

the histogram in combination with a histogram cache at the individual nodes. A direct mapping would require every node to maintain the complete list of participating nodes and also the mapping of the individual histogram buckets to the nodes. Using the key-based routing, each node stores only $O(\log n)$ neighbors and the mapping is done automatically by the underlying fabric. Updating a histogram via a distributed broadcast is not more expensive than distributing an updated histogram from a central site. Furthermore, we reuse functionality already implemented by the Peer-to-Peer (P2P) substrate and leverage the increased flexibility and the automatic handling of node failures. In Chapter 4, we present a detailed comparison between several mapping strategies with regards to data load balancing and query locality.

Node Arrival

When a node joins the HiSbase network, the active histogram will be transmitted to that node and the node needs to receive the data according to its responsibilities. For this purpose, HiSbase reuses the mechanisms of the DHT structure to determine the arrival of new nodes. In Pastry (Rowstron and Druschel, 2001), nodes are notified if the leaf set (the nodes which have similar identifiers) changes. Algorithm 2.4 describes how a notified node determines the data it is no longer responsible for. For this purpose, it compares its set of regions before and after the notification. The node then redistributes the moveable data and the newly joined node updates its database.

Node Departure

HiSbase is developed for an environment where the participating servers are quite reliable. High churn is currently not in our focus as distributing the envisioned amounts of data across unreliable nodes is not very useful. Nonetheless, some nodes might temporarily fail. As mentioned in the introduction of this chapter, HiSbase does not *replace* but *complement* the "traditional" data centers since these also serve as data sources for distributing the data in HiSbase. If a node leaves the network its direct neighbor nodes take over part of its data. The neighboring nodes refetch that data from the appropriate archives.

2.2.3 Data Distribution (Feeding)

Connected data centers directly feed data into HiSbase as illustrated by Figure 2.1. In HiSbase, the histogram is used to determine how to allocate data on nodes. All nodes maintain the data objects which are in their histogram buckets, independently from the archive the data comes from. HiSbase abstracts from the specific database system, which allows the use and evaluation of various traditional as well as main memory database systems.

Data archives that want to publish their data in HiSbase connect to any HiSbase node, preferably to a node nearby or to a node that has a high network bandwidth. Proceeding according to Algorithm 2.2, the contacted node uses the *lookupPoint* method of its histogram to locate the histogram bucket that contains a data object. Then it routes the object to the DHT identifier of this region. The message contains the data object and information about the data source. Via the underlying DHT mechanism, the data item arrives at the responsible node, which updates its database.

Distributing each data item individually results in a very high overhead. The precomputed histogram allows us to optimize the feeding stage by introducing *bulk feeding*. A node that feeds data into the network buffers multiple objects for the same region until a threshold is reached. Time-based as well as count-based thresholds are applicable.

Integrating new data sets is achieved by feeding them into the network as described above after the according tables are created at each node. If the new data set is a detailed survey of a sky region that has not yet been covered by any existing archive in the community network, it might be appropriate to create a new histogram in order to improve the data load balancing (Section 2.2.5). In that case, a data sample of the survey is extracted and integrated into the training phase. Chapter 5 discusses data feeding in more detail.

2.2.4 Query Processing

Region-based queries are submitted to any node of the HiSbase network. The node extracts the multi-dimensional area A from the query predicate. It selects an arbitrary identifier r_c from the set of intersecting regions, which is determined by *lookupArea*. The node p_c which is responsible for region r_c is the *coordinator*. The coordinator collects intermediate results and performs post-processing tasks (e. g., duplicate elimination).

Let us assume a region-based query was issued at node d in Figure 2.4. The area of the query is marked with the thick-lined rectangle in Figure 2.3(a). Thus, relevant to our example query are regions 1 and 3. If node d covers regions relevant to the query, it becomes the coordinator itself. This is not the case in our example. We select region 1 as r_c and thus node a becomes the coordinator. Node d forwards a coordination request to node a. The coordination request contains the query and the relevant regions. After node a receives the coordination request, it issues the query to its own database (as it covers relevant regions) and sends the query to all other relevant regions. Node b also participates in the query processing in our example as it covers region 3. It sends its intermediate results back to the coordinator, node a. After having received all intermediate results, node a returns the complete result to node d.

Nodes may cover several regions. As region identifiers are used for submitting queries, nodes can receive the same query several times. Each node stores a hash of currently processed queries to avoid multiple evaluations of the same query. Results and error messages are directly transmitted to the coordinator or the submitting node without using the overlay routing algorithm. Chapter 6 discusses more details on query processing and query coordination within community-driven data grids and presents the evaluation results from our query throughput measurements.

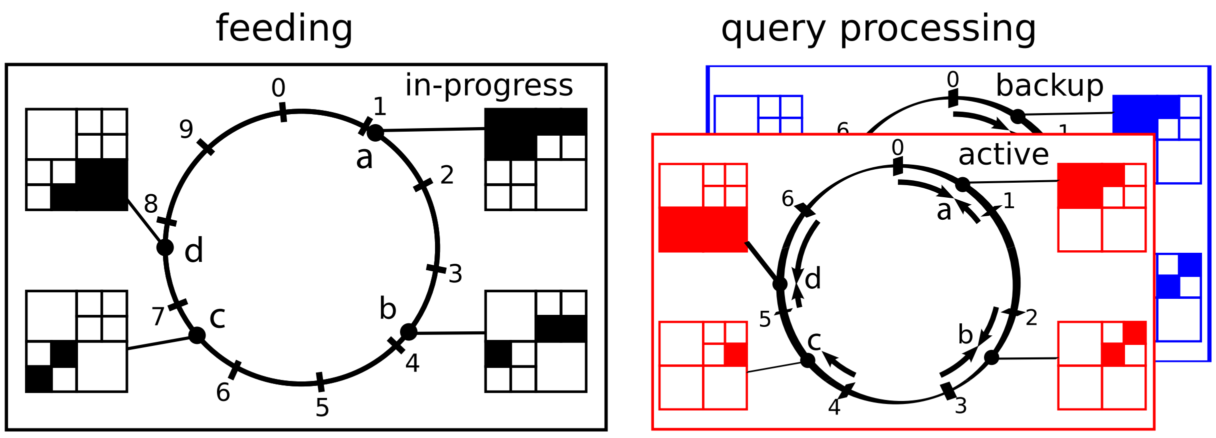


Figure 2.5: Histogram evolution

2.2.5 Query Load Balancing

There are several techniques for combining our data load balancing approach described so far with query load balancing techniques to efficiently handle query hot spots. In order to achieve this, we extend the use of our training phase and employ techniques that redistribute load at runtime.

We enhance the training phase with query statistics such as earlier workloads. Based on these statistics, the data partitioning can be modified to enable the application of query load balancing techniques such as replication or load migration. For a detailed discussion about this workload-aware data partitionings, we refer the reader to Chapter 7.

Using two parallel Pastry rings with different histograms increases the data availability within the HiSbase network. By changing the offset (or the space filling curve) of the mapping process from Section 2.2.2, the second histogram stores the data on different nodes and both copies are available during query processing.

We also introduce a *master-slave* hierarchy, where idle nodes can support overloaded nodes by offering their storage and compute resources. These may be necessary to cope with short-term changes in query load distribution. Whether a node is overloaded or constitutes a potential slave-node is determined based on workload statistics collected during run-time. These statistics can also augment the training phase for the next histogram evolution.

2.2.6 Evolving the Histogram

The histogram serves HiSbase as a partitioning function, defining the data set which a node is responsible for. HiSbase nodes maintain three histograms and their accompanying data sets to improve load balancing or level long-term data shifts. From our perspective, three data copies offer a good data availability at a reasonable management overhead for e-science scenarios. Each pair of histogram and data set can evolve during the lifetime of a HiSbase instance and has one of the following three functionalities: the *in-progress*, *active*, and *passive* functionality.

in-progress The currently running *feeding* process, which is described above, distributes data according to the in-progress histogram. After a new histogram has been distributed, the HiSbase nodes build this in-progress data set and store it on disk.

active Once the build-up phase of the in-progress histogram is completed, they become the active histogram and data set. Both are used during *query processing* and nodes keep them completely (or at least the relevant parts) in main memory. Furthermore, HiSbase nodes use the active histogram for messaging.

passive The completely updated data set is additionally kept on disk as backup for the active data set. This preserves the active data set beyond the lifetime of the current network and can be used if a node is restarted with the same identifier.

Figure 2.5 illustrates a scenario where the in-progress histogram contains additional regions while the active and passive histograms are the same as in Figure 2.4.

Any of the participating nodes can be used to inject an updated version of a histogram by broadcasting it to the HiSbase network. Our concepts for query load balancing at runtime are discussed in Chapter 8.

2.2.7 HiSbase Evaluation

We use three different evaluation settings in order to measure the features of community-driven data grids: a set of tools to analyze various characteristics of partitioning schemes, HiSbase instances running within an overlay network simulator, and deployments in various test beds.

The analysis of the individual partitioning scheme characteristics provides us with valuable insights for comparing and choosing different candidate data structures. Simulated instances allow us to explore systematically the network flow or communication patters under various conditions. Finally, the distributed instances are the key to test our system in a realistic environment and to evaluate the merits for scientific users.

Partitioning Schemes

Although many e-science communities require a scalable data management, they mostly have slightly different types of analysis tasks and therefore a wide range of requirements for the data management infrastructures. The scientific researchers require support and simple tools that help them to find an optimal partitioning scheme for their particular interest. Our testing framework developed within the HiSbase project enables the researchers to compare various candidate partitioning schemes based on several properties and to choose the one partitioning scheme which fits best their needs. Among these tools is a graphical user interface (GUI, shown in Figure 2.6) which allows for comparing different histogram data structures. Moreover, the GUI supports query submission and query analysis and shows status information on the connected HiSbase nodes.

The FreePastry Library

For the implementation of our prototype we use FreePastry[1], the open source Java-implementation of Pastry (Rowstron and Druschel, 2001), currently maintained by the Max-Planck-Institut for Software Systems. FreePastry provides the underlying key-based routing fabric and P2P-based multicast communication (i. e., Scribe by Castro et al., 2002).

FreePastry provides an implementation of the Common API (Dabek et al., 2003), which describes a common interface for DHT-based implementations. During our implementation, we aimed at programming only against those interfaces in order retain as much independence from the underlying overlay network implementation as possible.

[1]http://freepastry.org/

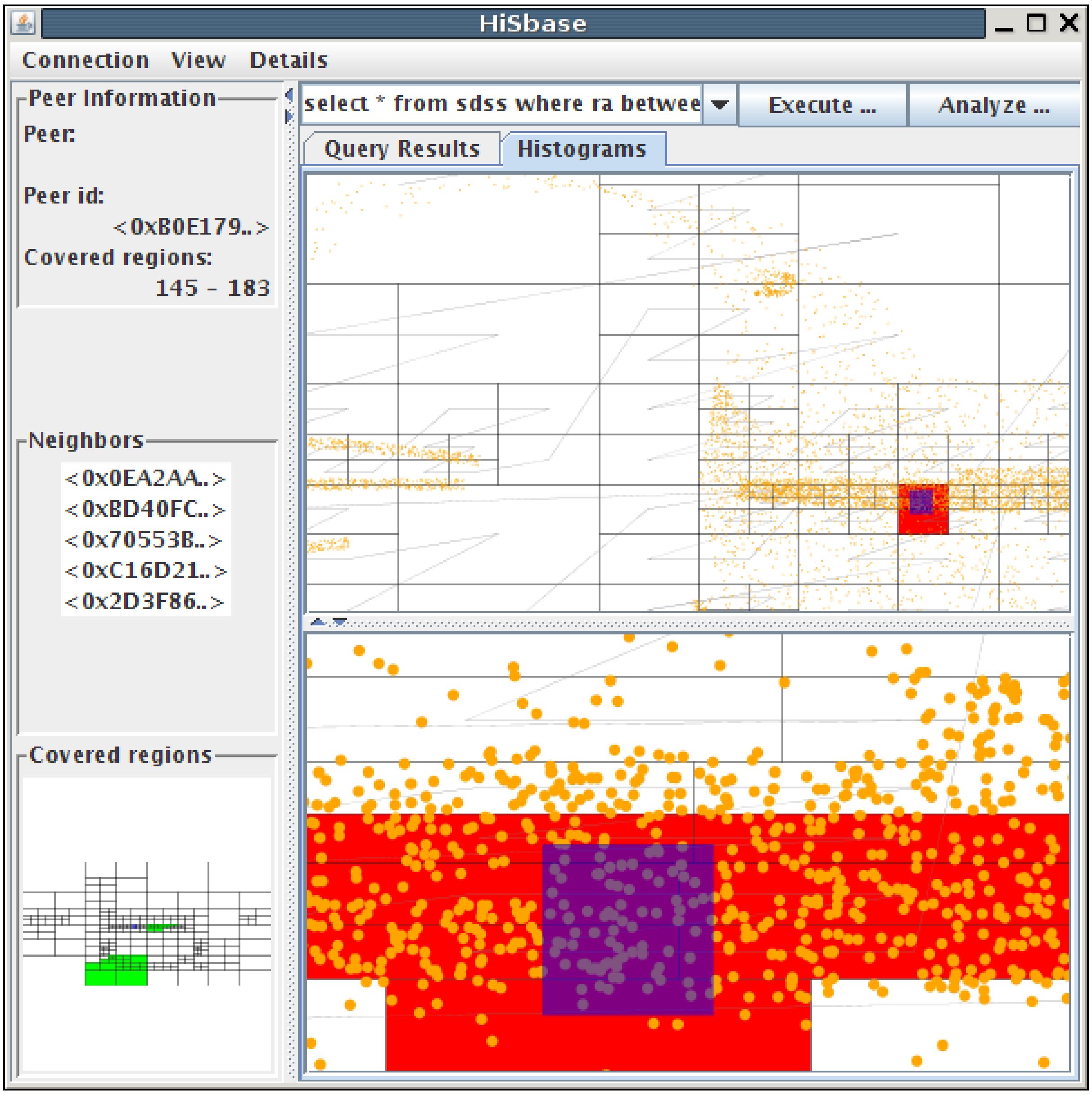
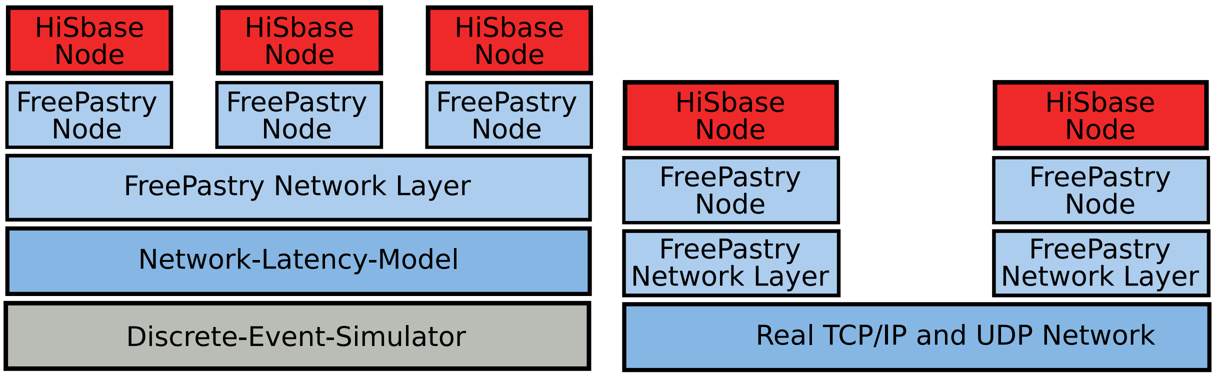


Figure 2.6: The HiSbase GUI



(a) FreePastry Simulator **(b)** Distributed FreePastry

Figure 2.7: Simulated and distributed evaluation environments on FreePastry

Simulation Environment

Moreover, FreePastry provides enough abstraction from the underlying network layers, in order to use our application unmodified for both simulation and distributed deployments. Reusing the same code in both environments was one of our major incentives to favor this simulator over other prominent simulators such as ns-2[1]. In Figure 2.7(a), we give an coarse overview of the simulator environment within FreePastry. The simulator uses discrete events and thus allows non-linear execution to speed-up simulations considerably. It also provides a module with various topologies to model network latency, e. g., Euclidean or spherical networks or explicit latency matrices. Above this layer, FreePastry has its network layer and allows to run several thousand nodes within a single Java virtual machine. The simulator does not model network bandwidth, message loss, nor varying latency due to congested network resources. As HiSbase aims at grid-based community infrastructures having dedicated resources and we also perform evaluations in real deployments, the benefits of using a single code base outweighs the missing features of the simulator.

Distributed Instances

Due to the simplifications within our simulation environment, we consider it as very important to deploy our prototype also in real test beds. Tests in real deployments exemplify actual benefits for the research communities. Figure 2.7(b) shows the communication layers for the distributed scenario.

We deployed HiSbase on several nodes within our lab. Measurements with these nodes show the performance using high bandwidth networks and low latency within a single institution. Our measurements using nodes of the AstroGrid-D test bed represent the performance of our community-driven approach within a nation-wide data grid using high-bandwidth networks to interconnect dedicated, powerful resources. Finally, we used PlanetLab[2] for performing experiments. PlanetLab is a test bed for distributed applications. Though the test bed rather targets projects that evaluate distributed algorithms or protocols, we integrated several PlanetLab nodes into a network with our AstroGrid-D resources, reaching up to one hundred nodes.

[1]http://nsnam.isi.edu/nsnam/index.php/Main_Page
[2]http://planet-lab.org/

2.3 Related Work

The HiSbase approach provides several benefits to e-science communities by addressing domain-specific data and query characteristics. HiSbase offers high throughput via parallelization, higher cache locality, and load balancing across several sites compared to centralized data management. HiSbase enables scalable sharing of decentralized resources within a community as it uses the DHT mechanism of key-based routing for data distribution and message routing. Using these techniques, new nodes can be easily added to the network and heterogeneous database management systems can be integrated with little effort as each HiSbase node only maintains its own local database configuration.

In the following, we present related work from areas such as distributed databases, P2P architectures, and scientific data management.

2.3.1 Distributed and Parallel Databases

Using parallelism and data partitioning to increase query throughput are well-established techniques from distributed and parallel databases which motivated us to use them as pillars for the architectural design of community-driven data grids. For example, Özsu and Valduriez (1999) describe in depth the general concepts and algorithms for distributed databases.

Kossmann (2000) provides a detailed survey of query processing techniques within distributed systems. Intelligent query processing and well-designed data placement are key techniques for realizing scalable data management solutions such as an information economy (Braumandl et al., 2003).

The field of parallel databases (e. g., Abdelguerfi and Wong, 1998) has brought up valuable insights in the area of query parallelization and infrastructure designs. A shared-nothing approach (DeWitt and Gray, 1992) where each node has an individual data storage and nodes communicate only via a shared network is considered the most scalable technique.

Compared to HiSbase, distributed databases run in a more homogeneous setting whereas parallel databases are not designed for world-wide distributed resources. Autonomous database systems (Pentaris and Ioannidis, 2006) also deal with the correlation of several data sources. However, data is not distributed across participating servers (adhering to the nodes' *autonomy*) and thus correlation needs to be done at the client sites which leads to additional data traffic.

Another important aspects induced by the vast number of distributed data sources are heterogeneity and provenance. Recently, a new pay-as-you-go approach (Salles et al., 2007) within so-called dataspaces (Franklin et al., 2005) was identified. As opposed to data integration systems (Naumann et al., 2006; Rahm and Bernstein, 2001), data co-existence is possible in dataspaces. Monitoring data provenance becomes increasingly important within distributed systems integrating data from various sources on demand, e. g., to ensure reproducibility of results. Recent work on provenance in databases includes, for example, Buneman and Tan (2007) and Davidson et al. (2007). A further treatment of data integration, dataspaces, or provenance is beyond the scope of this thesis. We therefore assume that data being fed into HiSbase either adheres to a common schema or has already been transformed properly.

2.3.2 P2P architectures

DHT architectures such as CAN (Ratnasamy et al., 2001), Chord (Stoica et al., 2001), Pastry (Rowstron and Druschel, 2001), and Tapestry (Zhao et al., 2004) overcome the limitations of centralized information systems by storing data in a distributed one-dimensional key space (except for CAN which uses a d-dimensional torus). While these systems achieve load balancing by randomly hashing data and peers to their key space, they neither support multi-dimensional range queries nor preserve spatial locality. HiSbase work is reminiscent of the achievements in P2P-based query processing (Huebsch et al., 2003).

Instead of DHTs, other proposals build distributed tree-based structures that already incorporate range query capabilities for one-dimensional data. For example, Jagadish et al. (2005) describe a distributed balanced binary tree, BATON. If the target node of a message is not within the subtree of the sender, the message is routed towards the root of the tree. In order to reduce the routing overhead on the nodes close to the root of the tree, BATON also builds "vertical" routing paths. Ranges can be queried by seeking the start of the range and then perform an in-order traversal until the range is completely processed. P-Grid (Aberer et al., 2003) uses a trie-based infrastructure and performs routing along these prefixes. Besides additional support for replication, Datta et al. (2005) describe a "shower" algorithm on P-Grid in order to trade an improved response time for range queries for more messages.

A large variety of systems have been proposed to augment DHTs in order to support (multi-dimensional) range queries (Banaei-Kashani and Shahabi, 2004; Ganesan et al., 2004b; Shu et al., 2005; Tanin et al., 2007) or to address data (or execution) load balancing in P2P environments (Aspnes et al., 2004; Crainiceanu et al., 2007; Ganesan et al., 2004a; Pitoura et al., 2006). These systems are predominantly designed for settings that are very dynamic, i. e., data hot spots and the data itself change very frequently and the systems have a very high churn. This flexibility comes at the price of dealing with each data object (of several hundred million data objects) individually. We exemplify some of these systems below and discuss how they relate to HiSbase.

One approach (Banaei-Kashani and Shahabi, 2004) uses Voronoi diagrams in order to partition the data space and to support queries on multi-dimensional data. Independently, the MURK system (Ganesan et al., 2004b) uses k-d trees to realize a similar idea. In these systems, peers covering large data partitions have more neighbors, while in HiSbase the number of neighbors is independent from the number and size of covered regions. SCRAP (Ganesan et al., 2004b) directly applies a space filling curve to the data and assigns one-dimensional ranges to peers. In HiSbase, the submitting peer exactly determines the histogram regions in the multi-dimensional data space and only these peers are contacted during query processing while SCRAP can only approximate a multi-dimensional query range using multiple one-dimensional ranges.

The distributed quadtree index (Tanin et al., 2007) is a distributed data structure for objects with multi-dimensional extents and supports range queries. Each quadtree node is represented by its centroid and these are randomly placed on the key space of an underlying DHT structure (e. g., Chord). Two levels of the distributed quadtrees can be configured to limit where objects are stored: one level (f_{min}) defines the minimum depth, the other level (f_{max}) defines the maximum depth. Thus, distributed quadtrees aim both at avoiding the bottleneck of congested nodes in layers above layer f_{min} and too much fragmentation by storing objects in layers below layer f_{max}. When HiSbase applies the Z-quadtree as partitioning scheme it only maps the leaves to the key space and does this equidistantly according to a space filling curve. Thus a peer covers data from neighboring regions which then can be stored in the same database with a higher probability than in the distributed quadtree. Each peer within the distributed quadtree caches direct links to the children of the quadtree nodes it is covering. Thus, it takes $O(\log n)$ hops to find an f_{min}-node and then a constant number of steps to reach the relevant leaves. These steps also have to be processed with data objects without an extent which are stored at level f_{max}.

In HiSbase, no additional routing steps are necessary. HiSbase discovers the relevant region directly and routes to the responsible peer using $O(\log n)$ messages.

Shu et al. (2005) describe an online balancing algorithm for frequent changes in data hot spots which is also based on quadtrees. The quadtree leaves are mapped on a skip graph (Aspnes and Shah, 2003) layer using a space filling curve. Aiming at data sets with high update rates, the authors devise algorithms that require an initial phase such as the training phase of HiSbase and each peer only needs partial knowledge of the complete data distribution. However, peers are only allowed to manage regions on the same Z-level while in HiSbase there is no such restriction. Accounting for the rather stable data sets of e-science communities, these communities benefit more from techniques increasing the query throughput of data management infrastructures than from such an approach.

How to achieve load balancing in one-dimensional, range-partitioned data is described in (Aspnes et al., 2004; Ganesan et al., 2004a). Ganesan et al. (2004a) show that load balancing schemes for range-partitioned data in highly dynamic P2P networks either need to adjust the load between neighbors or need to change peer positions within the range. SCRAP is an extension of (Ganesan et al., 2004a) to multi-dimensional data. Aspnes et al. (2004) only maintain representative values of the data ranges in the skip graph. Load balancing between these data ranges is achieved by arranging less-filled (open) buckets close to full (closed) buckets. HotRod (Pitoura et al., 2006) addresses query hot spots on one-dimensional data by replicating popular data ranges on additional rings. Data is stored on the DHT using order preserving hash functions and as soon as access statistics show that a peer is increasingly accessed HotRod replicates (hot) data to other virtual rings. While HotRod determines the ranges to be replicated during runtime, HiSbase has opted for replicating the partitioning scheme (the histogram) to all participating nodes, as the overall data distribution is fairly stable. Moreover, HiSbase allows a high flexibility regarding the actual histogram used by a particular community.

P-Ring (Crainiceanu et al., 2007) approaches data skew in an orthogonal manner in comparison to HiSbase. While HiSbase adapts the buckets of the histogram data structure to data skew and distributes these across the cooperating peers, P-Ring has the notion of “helper peers” that support peers which are overloaded by skewed insertions either by data redistribution between neighbors or by merging their data into a neighbor’s range. Considering multi-dimensional range queries, P-Ring would need to approximate the query area with multiple one-dimensional intervals. Using the insertion rate of 4 data items per second as in the simulation study of P-Ring, importing 80 million objects would last 33 weeks (20 million seconds), which is inappropriate for e-science communities having terabyte-scale data sets.

2.3.3 Scientific and Grid-based Data Management

Within many scientific communities, data management challenges propelled and are still triggering many innovative ideas and technologies in order to ease the day-to-day experience of the researchers. The D-Grid initiative accommodates several community-driven efforts in order to build scalable grid-based infrastructures in various research areas. The research-specific services offered by the individual communities range from user-friendly secure grid access in medical applications by MediGRID (Krefting et al., 2009) to a collaborative data and processing grid for the climate community provided by C3Grid (Kindermann et al., 2007). Together with AstroGrid-D, these communities have also identified synergies within the individual data management services (Plantikow et al., 2009). Other groups within the framework of D-Grid also focus on security (Gietz et al., 2009) and VO management (Kirchler et al., 2008) aspects which are important to ensure the acceptance of grid-based solutions within the scientific community. Several areas have been identified (Foster and Iamnitchi, 2003; Ledlie et al., 2003) where P2P technologies and grid computing can be combined in order to provide scalable infrastructures. We agree that this combination indeed fosters interesting options for the data management design and, with this in mind, we designed community-driven data grids accordingly. In the following, we describe other proposals for scientific data management.

MAAN (Cai et al., 2004) adds multi-attribute and range-queries to grid information systems. While string-attributes are still randomly hashed on the key space, they use a locality preserving hash function (and if the distribution function is known, a *uniform locality preserving* hash function) for numerical values. Query processing is done in an iterative way (by querying each attribute individually, or using a single-attribute dominated query processing algorithm). According to each attribute, the data objects are inserted. So if a data object consists of 140 attributes, it is stored 140 times in the overlay network. This amount of redundancy is prohibitive for large multi-attribute e-science data sets. HiSbase additionally accelerates query processing by contacting the relevant nodes in parallel.

VoroNet (Beaumont et al., 2007) creates a P2P overlay network based on Voronoi tessellations of high dimensional data spaces and offers poly-logarithmical routing performance with regards to the data objects. It builds the overlay network between the data objects which store interconnections to objects close-by (with regard to a distance metric) and a far-distant in order to get short-cuts for routing. It is required that all nodes know the total number of objects for which the system was optimized in order for routing algorithms to function properly. Although they can handle data skew via the Voronoi tessellations, data remains on the publishing nodes which can result in an imbalance, e. g., if one node shares more data with the community than others. The experiments were conducted with up to 300 000 two-dimensional objects from both skewed and uniform distributions. Their flexibility in changing the tessellation comes at the price that each data object is treated individually. It is unclear how the approach scales with several millions of objects and if the expected number of objects is guessed wrong.

GIME (Zimmermann et al., 2006) takes a different approach to geotechnical information management in federated data grids. The system adheres to the data autonomy of the participating institutions and uses a replicated index (based on quadtrees or R-trees) for managing the bounding boxes of participating archives. Thus, it reduces the number of messages by submitting the query only to such archives whose minimum bounding box actually intersects the query area. Load imbalance can arise for data archives covering a large area. These archives have to process more queries than small archives and several data sets cannot be combined directly on-site. AstroPortal (Raicu et al., 2006) combines digital images from several astronomical archives by offering a grid-based stacking service in order to create a “complete picture”.

OGSA-DAI is widely used as data access interface in several Grid communities world wide. Therefore, OGSA-DAI would be the perfect candidate to offer Grid-based access to HiSbase in a production system. Kottha et al. (2006) have conducted a performance evaluation of OGSA-DAI within the MediGrid project for medical applications.

Several inspiring ideas with regards to scientific data management have been proposed in the context of a technique called Bypass-Yield Caching (Malik et al., 2005). The goal of this technique is to cache results from federated scientific archives close to the user in order to reduce network traffic and shorten response times. Based on the estimated result set size of a query the cache decides whether to cache a result. The result size estimation uses query templates (Malik et al., 2006) extracted from the query history. For selectivity estimations, Malik and Burns (2008) propose a technique which exploits the workload information and query feedback by applying a recursive least square algorithm in order to minimize the estimation errors. When the scientific partners retain data autonomy, a distributed join has to be scheduled across several sites. In order to increase the throughput in such scenarios, Wang et al. (2007) propose algorithms based on spanning tree approximations for optimizing the scheduling of the distributed joins. It is an interesting issue for future research to combine their caching techniques with our throughput optimizing infrastructure and to evaluate synergies.

CHAPTER 3

Community Training: Selecting Partitioning Schemes

This chapter provides an in-depth discussion of the training phase, where the partitioning scheme for a community-driven data grid is created (Section 3.1). We discuss a variety of candidate data structures for such a partitioning scheme, ranging from traditional multi-dimensional indexes to application domain-specific data structures (Section 3.2). In Section 3.3, we describe several criteria for comparing such partitioning data structures and apply these criteria during the evaluation of the variants described in Section 3.2. Related work (Section 3.4) and a summary (Section 3.5) conclude this chapter.

3.1 Training Phase

We use a training phase to create and distribute partitioning schemes that describe how data objects from various archives are to be partitioned. The training phase comprises three steps:

1. Extract the training samples,
2. Create the partitioning scheme, and
3. Distribute data according to the partitioning scheme.

The training samples are representative subsets from all data archives that are to be distributed within the network. Given the size of existing and anticipated data sources (several terabytes each), it is not feasible to perform the training on the complete data sets. We use functionality provided by relational database systems to extract random samples. Based on these training samples, we build the partitioning scheme. The created partitioning scheme is then evaluated considering the identified application-specific data and query properties. After having selected a partitioning scheme, we create the individual *data partitions* from the participating data archives. These data partitions are then distributed to the shared resources within the data grid, e. g., using GridFTP or our data feeding techniques from Section 5. At the resources, the data is loaded into a database and made accessible via the middleware infrastructure.

Quadtrees and the *zones* index are the two data structures we compare in the following with regard to their fitness for being used as a partitioning scheme for community-driven data grids in astrophysics.

3.2 Data Structures

In quadtree-based partitioning schemes, the partitions correspond to the leaves of a quadtree. *Quadtrees* (Finkel and Bentley, 1974; Samet, 1990) are a well-known spatial data structure and use the principle of recursive decomposition to partition a d-dimensional data space. Quadtrees are recursively defined to be either a leaf with a d-dimensional hypercube data region or an inner node with 2^d subtrees. In a 2-dimensional data space, an inner node has four children (quadrants) that cover equal-sized convex data regions.

In particular, communities having skewed data sets can benefit from the capability of quadtrees to adapt to the data distribution. Sparsely populated areas of the data space are represented by leaves covering a large data region and densely populated areas are partitioned into several leaves covering small areas. Thus, the amount of data within each leaf, and therefore within each partition, can be held approximately equal. In very pathological cases, however, where data is concentrated in a very small area, quadtrees degenerate to a tree having many empty leaves. Partitioning schemes without empty leaves are preferable in our setting because they allow us to directly map data partitions to shared resources.

One approach to address the issue of empty leaves is a *median-based heuristics* that splits a leaf at the median instead of at the center. For our astronomical example, the heuristics determines the split point (m_{ra}, m_{dec}) by computing the median for *ra*-coordinates and *dec*-coordinates independently. Our heuristics is similar to the technique used by *optimized point quadtrees* (Finkel and Bentley, 1974), which only compute the median in the first dimension and thus guarantee that no leaf contains more than half the data of the original leaf. Our heuristics, which computes the median in all dimensions independently, offers a better data distribution in the average case. Figures 3.1(a) and 3.1(b) show a quadtree with regular decomposition and a quadtree using our median heuristics, respectively, both built during our evaluation.

Quadtrees can be created either top-down or bottom-up. Both approaches start with a single empty quadtree leaf. During top-down creation, the training sample is sequentially inserted into the leaf until a predefined threshold for the leaf capacity is reached. The leaf is split up and its data is distributed among its new subtrees. In the bottom-up approach, leaves have an unlimited capacity and all data is inserted into the initial leaf first. In the following, the biggest leaf is split in turn until a predefined number of partitions has been reached.

We prefer the bottom-up approach over the top-down approach for several reasons. Building quadtrees top-down requires to guess the leaf capacity in advance, which is hard for highly skewed data sets. Furthermore, building the partitioning scheme bottom-up is a requirement for other splitting strategies to work properly. For example, the median-based heuristics depends on all points within a leaf and would be inaccurate if calculated incrementally when the leaf threshold is reached. We furthermore consider it easier to give a rough estimate of the number of nodes participating in a network than providing a good guess for the leaf capacity.

The *zones* index (Gray et al., 2006) is an index structure developed in order to improve the performance of typical query patterns in astrophysics such as points-in-region queries, self-match queries, and cross-match queries. The principle behind the zones index is to divide the data space into zones of equal height h. The zone identifier of a point (*ra*, *dec*) is calculated by $floor((dec+90.0)/h)$ because the domain of the *declination* coordinate is $[-90.0, +90.0]$. Due

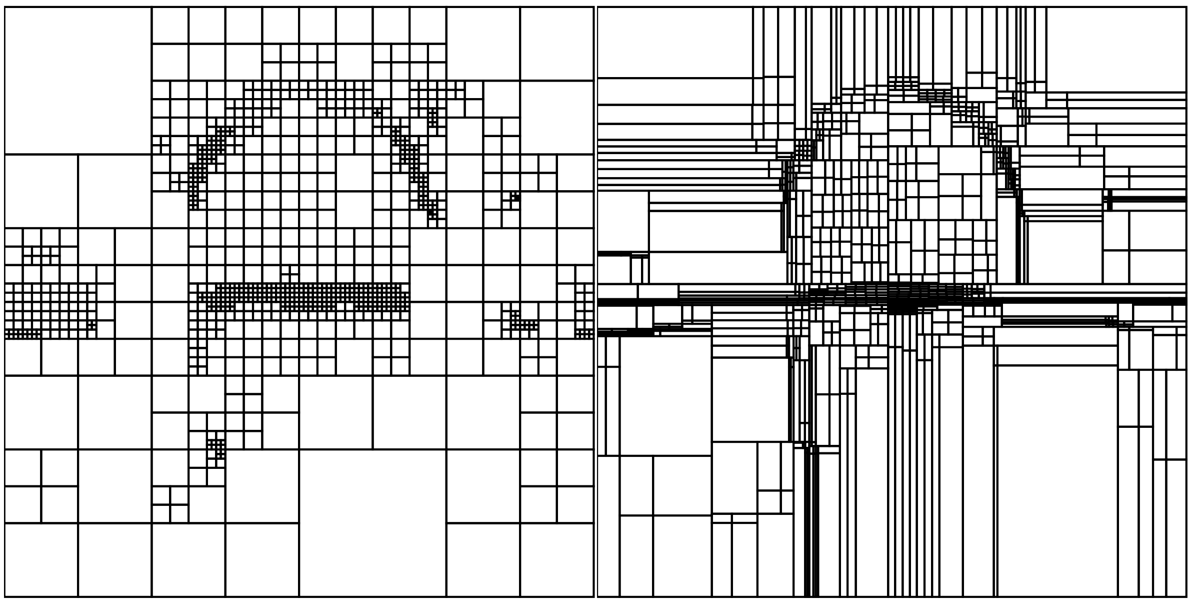

(a) without median heuristics **(b)** with median heuristics

Figure 3.1: Partitioning scheme with 1 024 partitions based on quadtrees (a) with regular decomposition and (b) with median heuristics

to their simplicity, zones are efficiently implemented directly in SQL and thus exhibit very good performance in tracking down relevant zones for a particular task. Applying the zones index to the algorithm for finding maximum-likelihood brightest cluster galaxies (Nieto-Santisteban et al., 2005) is a very good example for increased performance by implementing an algorithm as close as possible to the data: directly inside the database. The zones algorithm preserves spatial locality by grouping data elements from the same area in the sky into the same zone. Using zones as a clustered index, which also defines the physical layout of data besides accelerating data access, allows database optimizers to efficiently determine the query result.

We limit our discussion to these three data structures—the two variants of the quadtree and the zones—and refer the interested reader to the survey by Gaede and Günther (1998) or the book by Samet (2006) on multi-dimensional and metric data structures for more information on multi-dimensional access methods.

3.3 Evaluation of Partitioning Scheme Properties

We evaluated the fitness of the three different data structures described in the previous section by collecting several statistics. For the evaluation, we used our Java-based prototype of the HiSbase system, which also employs a training phase.

We drew training samples from the skewed observational data set P_{obs} and the uniformly distributed simulation data set P_{mil} provided by our cooperation partners and both described in the introduction (Section 1.2). The samples are roughly of equal size in order to specifically study the impact of data skew on the partitioning schemes. We varied the size s of the training samples to benchmark the quality of results obtained from small data samples. Finally, we generated partitioning schemes of different sizes n, with n varying from 16 (2^4) to 262 144 (2^{18}) partitions. If the partitioning scheme generates only non-empty partitions, we may generate as

Parameter	Value(s)	Description
P	P_{obs}, P_{mil}	Data set used for training sample extraction
s	0.01%, 0.1%, 1%, 10%	Size of the training set
n	$2^4, 2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}$, $2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}, 2^{17}, 2^{18}$	Size of the partitioning scheme

Table 3.1: Parameters for the training phase evaluation

many partitions as we have distributed grid servers available for distributing the data. If the construction method cannot guarantee the absence of empty partitions (which is generally the case), it is better to generate more partitions and assign multiple partitions to a server. We chose 2^{18} partitions as upper limit, since partitioning schemes at this scale offer us high flexibility with regards to transferring the partitions and gives a good ratio between number of partitions and number of nodes. We refrained from building a partitioning scheme, if the desired sizes would lead to splitting empty partitions due to a data sample with too little data. Therefore, as we will discuss in Section 3.3.5, we did not construct partitioning schemes with more than 16 384 (2^{14}) partitions from the 0.01% training sample and more than 131 172 (2^{17}) partitions from the 0.1% training sample, respectively, as this would have resulted in further splitting completely empty partitions. Table 3.1 summarizes the evaluation parameters.

During the training phase, we gathered the following information: 1) the duration, 2) the data distribution across partitions (measured in comparison to the partition storing the largest amount of data), 3) the variance in data population, 4) the number of empty partitions, 5) the differences among the results of training sets of varying size, and 6) the accuracy obtained during the training phase in comparison to calculating the partitioning schemes on the complete data set. The rationale behind investigating these characteristics is as follows.

1) Duration The duration[1] of the training phase provides a notion of how long it takes to get a good partitioning scheme. We observe that in many e-science communities, data sets are updated only every few months or on a yearly basis. Due to the invariance of the data, the training phase merely is a one-time cost.

2) Average data population The average data population in comparison to the biggest leaves gives a notion about how similar the data distribution is among the individual leaves.

3) Variance in partition population The variance is similar to the average data population. A low variance implies that all partitions contain approximately the same number of objects.

4) Empty partitions Without empty partitions, we can directly map data partitions to the shared servers. In cases where we cannot avoid empty leaves, we can create more partitions than the anticipated number of servers and assign multiple partitions to a server.

5) Size of the training set If outcomes of a small training set are comparable to the results of a large training set, we can use the smaller one and thus reduce the cost of the training phase.

[1]We performed the evaluation of the training phase on a Linux server equipped with four Intel Xeon processors at 2.33 GHz and 16 GB of RAM.

6) Baseline comparison Computing the histogram on the complete data set would generate the exact partitioning scheme. However, in many circumstances this is not feasible due to the enormous data volumes.

3.3.1 Duration

The graphs in Figure 3.2 show the duration of the training phase (in seconds) for both quadtree variants and for the zones on the different training sets for the 0.01%, 0.1%, 1%, and 10% training samples, respectively. The runtime of computing the zones is invariant to the data distribution and can be computed very fast by providing the partition cardinality. For partitioning schemes which consider data distribution, the duration increases with the number of partitions. The larger the training sample, the more the duration is dominated by the processing of the training sample. Therefore, the influence of a higher number of partitions is less prominent in Figure 3.2(d) compared to Figure 3.2(a)–(c).

For median-based quadtrees, the training phase lasts longer than for standard quadtrees. In addition to finding the biggest leaf, which is necessary in both cases, the median needs to be calculated. This is more expensive than only splitting a leaf. The duration increases similarly for both variants, as we use the $O(n)$ order statistics algorithm (Cormen et al., 2001) to compute the median instead of an $O(n \log n)$ sorting-based algorithm. The order statistics algorithm basically only continues partitioning that range which contains the desired element.

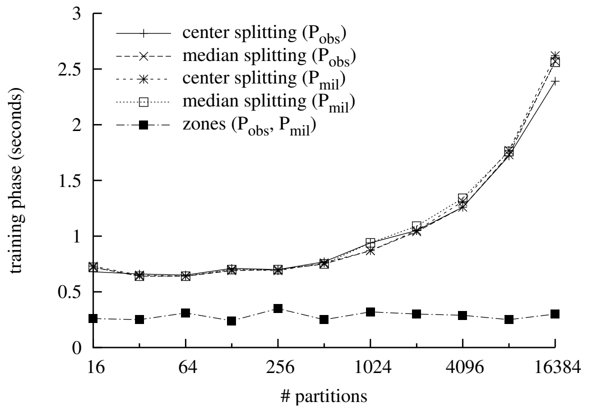
3.3.2 Average Data Population

The covered area of the data partitions can differ in size either due to the data distribution (quadtrees increase the resolution for highly populated data regions) or position (zones close to the poles cover a smaller area than regions close to the equator). When inserting the training samples into the data structures, each data object is assigned to a containing partition. We call the data objects that are assigned to a partition p, the *population* of p. Having defined the population of a partition, we now can compare the *average data population (ADP)* to the maximum population among the partitions. A more homogeneous data distribution results in a higher ADP. If the ADP is 100% then all partitions are equally populated.

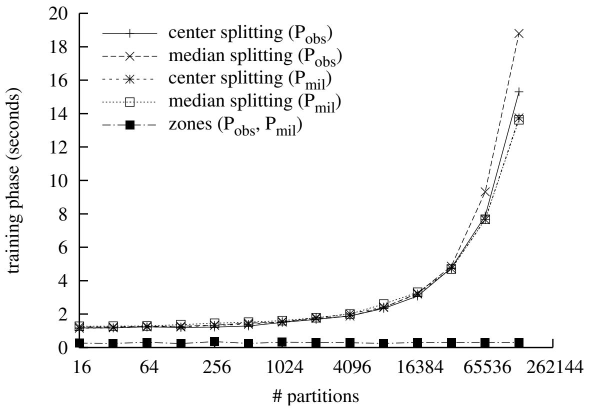
The ADP graphs for the quadtree variants for different training samples and the graph for the zones on the complete data set are shown in Figure 3.3. Constructing quadtrees bottom-up only considers the data distribution for selecting the most populated leaf. Therefore, results for all training sets are very similar. In conjunction with the median heuristics, quadtrees achieve a very good average population in both data sets, especially in P_{mil}. The significant drops of the ADP for median-based histograms on P_{mil} for uneven powers of 2 can be explained by the example shown in Figure 3.4. The figure shows the histograms for 4 096 (2^{12}), 8 192 (2^{13}), and 16 384 (2^{14}) partitions, respectively. In order to divide all 4 096 partitions equally, four times more, that is 16 384, leaves are necessary. Therefore, the partitioning algorithm is “still underway” during the uneven powers of two, such as 8 192 partitions. Zones achieve a good ADP in homogeneous environments, whereas in skewed data sets, the ADP is lower in comparison to the quadtree variants.

3.3.3 Variation in Data Distribution

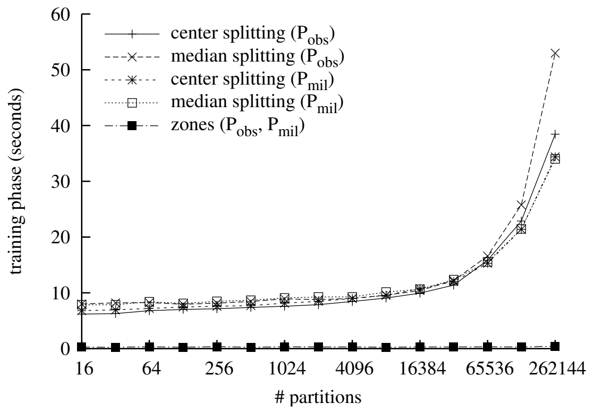
Next, we measured the standard deviation for each partition. The larger the deviation, the more diverse is the distribution of data points among the partitions. The graphs in Figure 3.5



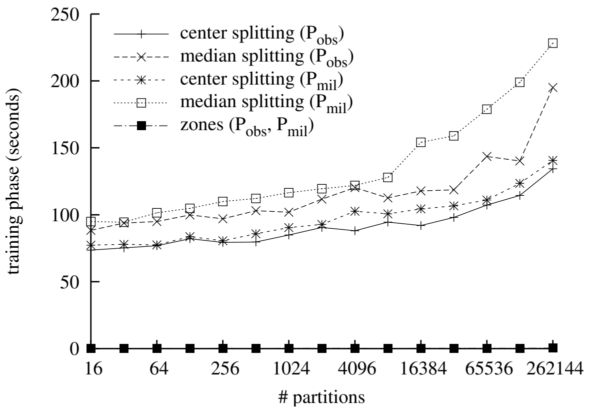
(a) 0.01% training sample



(b) 0.1% training sample

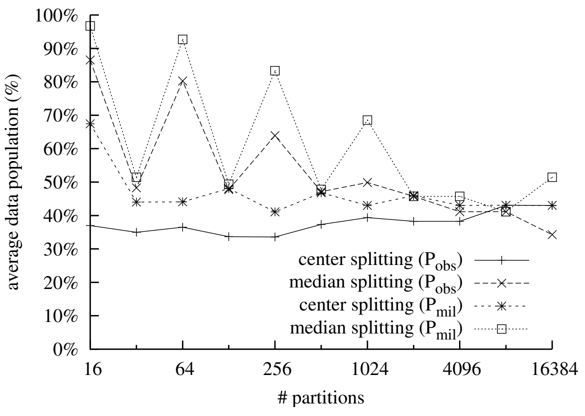


(c) 1% training sample

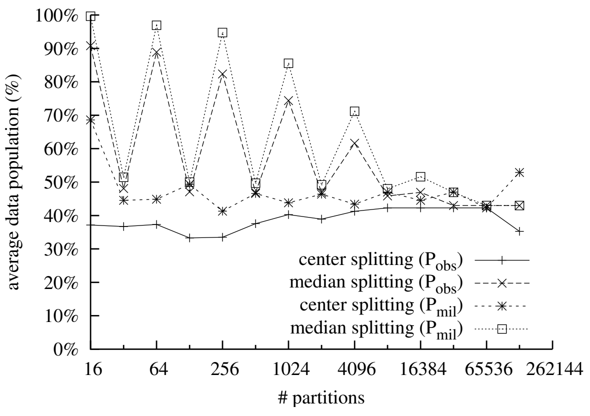


(d) 10% training sample

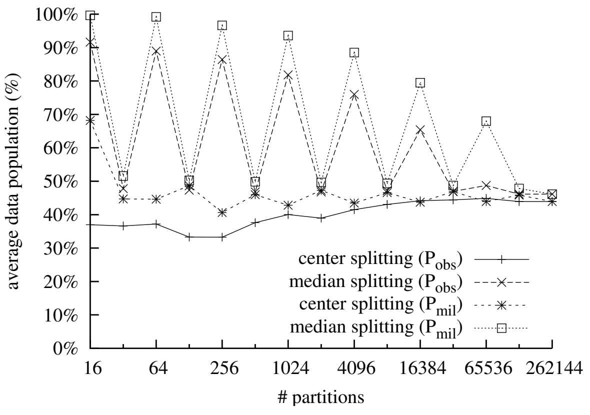
Figure 3.2: Duration of the training phase



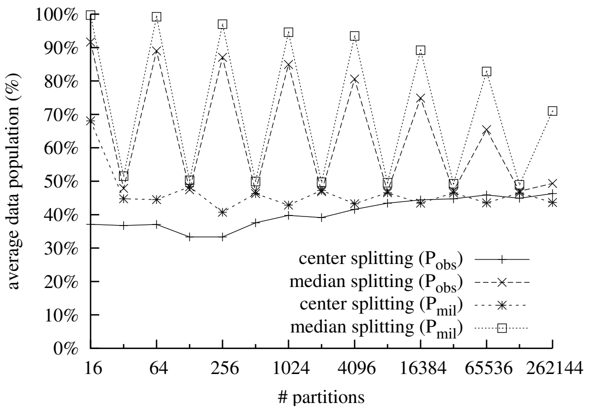
(a) 0.01% training sample



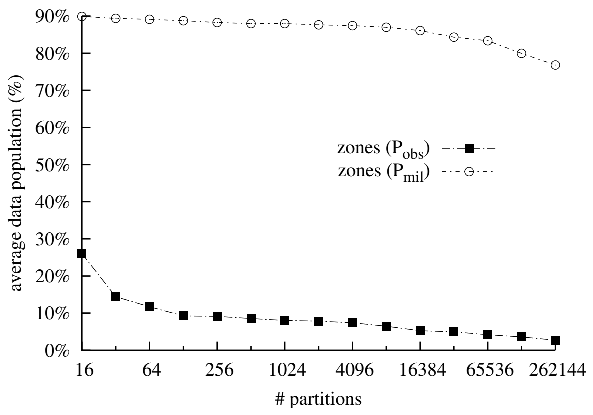
(b) 0.1% training sample



(c) 1% training sample



(d) 10% training sample



(e) Complete data

Figure 3.3: Average population of a partition in comparison to the partition with the highest population

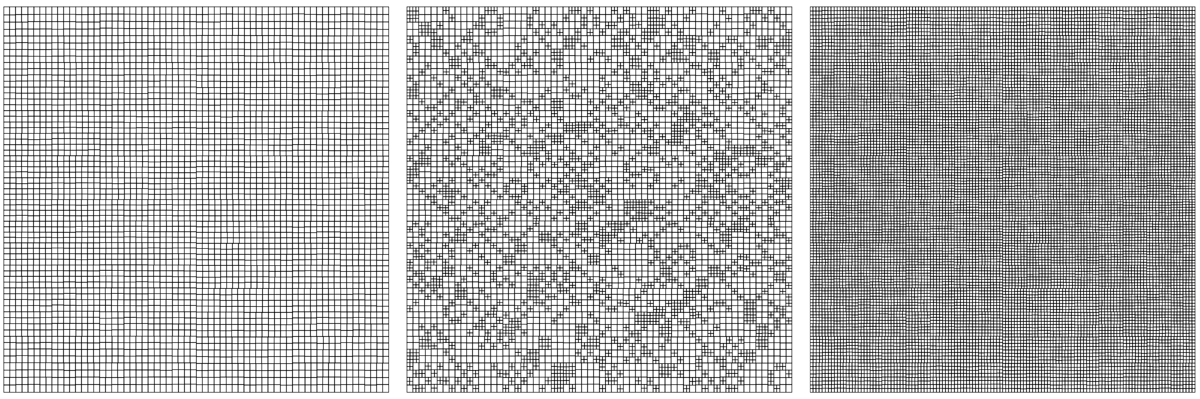
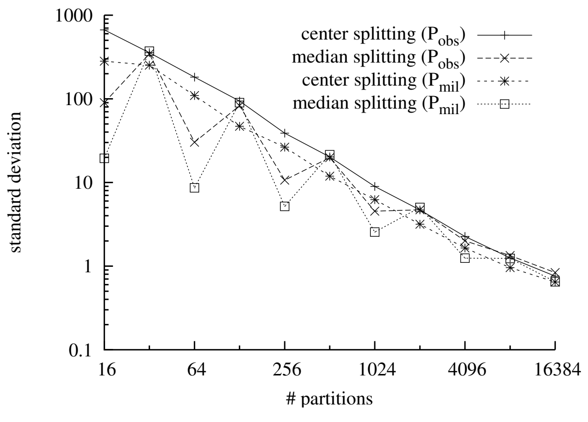


Figure 3.4: Median-based quadtree for P_{mil} with 2^{12}, 2^{13}, and 2^{14} partitions

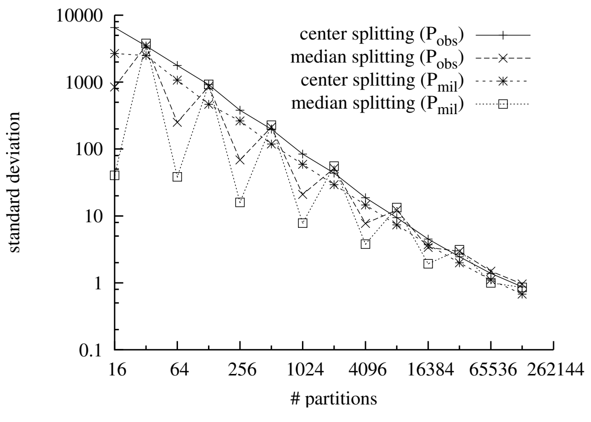
depict the standard deviation on a logarithmic scale. The median-based heuristics decreases the deviation for both data sets, especially for the uniform data. Generating 4 096 partitions from the 10% training sample results in a standard deviation of 1 432 and 92 for the median-based variant and of 1 827 and 446 for the standard quadtree for P_{obs} and P_{mil}, respectively. In each graph, the variance decreases because the size of the training set is kept constant and therefore the amount of data objects for each partition is reduced. The standard deviation of the zones approach (not shown) develops similar to the standard deviation of the quadtree. Due to the adoption to the data distribution, the deviation of the quadtree-based partitioning schemes is slightly (by a factor between 3 to 10) lower than the standard deviation of the zones.

3.3.4 Empty Partitions

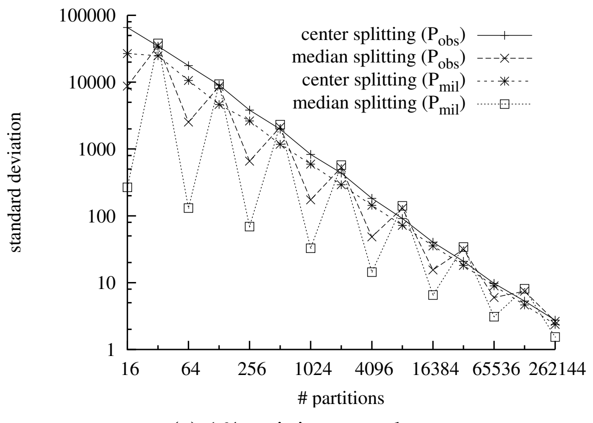
The next comparison deals with the number of empty partitions. The results are presented in Figure 3.6 as percentage of all partitions. While P_{obs} has large areas with no data (Figure 1.3(a) on page 5), P_{mil} does not. For both small samples (0.01% and 0.1%), however, the number of empty leaves increases significantly for both datasets from 16 384 partitions onwards (see Figure 3.6(a)). We discuss this deficiency of small data samples in the following section. For P_{obs}, all zones configurations have about 7% empty partitions. Roughly 13° of 180° at the bottom of Figure 1.3(b) contain no data, i. e., 7% of the declination domain are empty. As the zones divide the data space regularly, this ratio remains constant. For P_{mil}, no empty zone partitions exist. For the 1% and 10% training samples of P_{obs} (the latter is shown in Figure 3.6(b)), from 2 048 partitions onwards the percentage of empty partitions steadily decreases for the standard quadtree. The highest percentage of empty partitions is for the standard quadtree with 128 leaves. This is due to the fact that the percentage of additional spits that result in additional empty partitions is the highest when going from 64 to 128 partitions. From that on, the empty areas (the “withe space”) of our data sample have been roughly approximated and additional empty regions only appear at the “edges” of dense data areas, which explains our measurements for P_{obs}. The median heuristics completely eliminates empty regions. In Figure 3.6(b), we do not show the graphs for the samples of the uniform data set P_{mil} as all three data structures created no empty partitions if the training sample is sufficiently large.



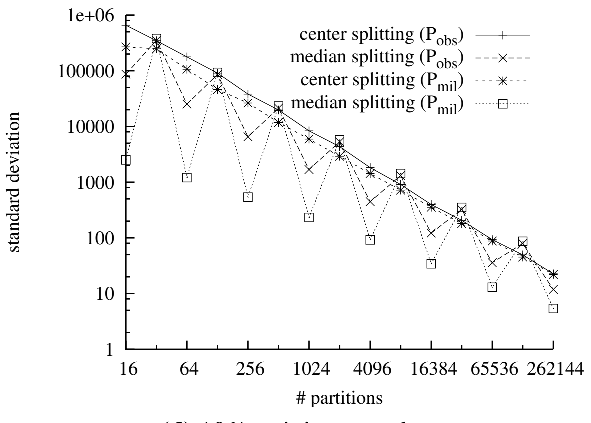
(a) 0.01% training sample



(b) 0.1% training sample

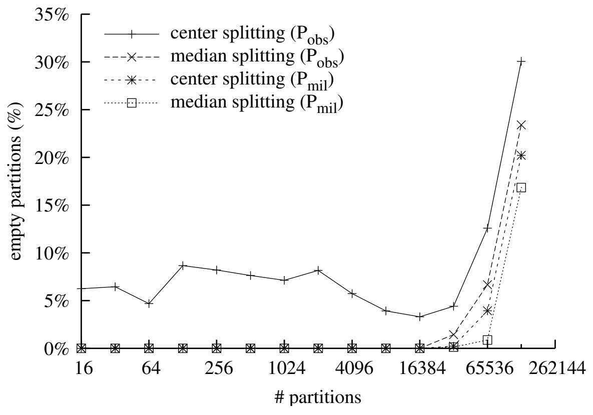


(c) 1% training sample

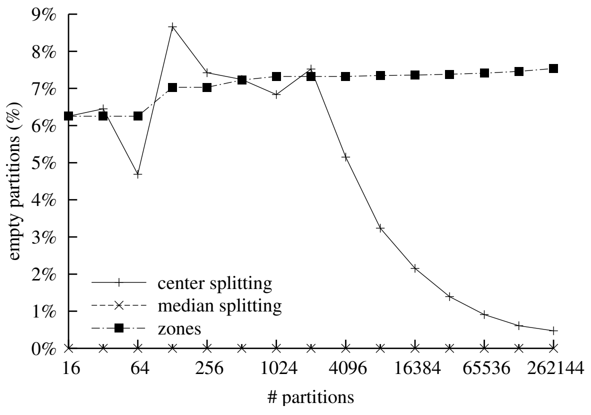


(d) 10% training sample

Figure 3.5: Variation in data distribution



(a) 0.1% training sample



(b) 10% training sample, P_{obs}

Figure 3.6: Empty partitions

Sample size	P_{obs}	P_{mil}
0.01%	14 103	16 851
0.1%	138 590	169 058
1%	1 382 150	1 692 883
10%	13 837 271	16 938 822

Table 3.2: Sizes of the used training samples

3.3.5 Size of the Training Set

We now discuss the effect of the sample size on the quality of quadtree-based partitioning schemes. To capture the quality of data samples for particular partitioning scheme sizes is helpful for the choice of the sample size. Table 3.2 shows the cardinality of the training samples for P_{obs} and P_{mil}, respectively. In the following, let the *sample ratio* $r_P^s(n) = \frac{|s\text{-\% sample of } P|}{n}$ be the number of objects per partition if the data is uniformly distributed, i. e., the size s of a training sample of a data set P divided by the number n of partitions. For example, the sample ratio $r_{obs}^{0.01}(1\,024) = 14\,103/1\,024 \approx 13.8$.

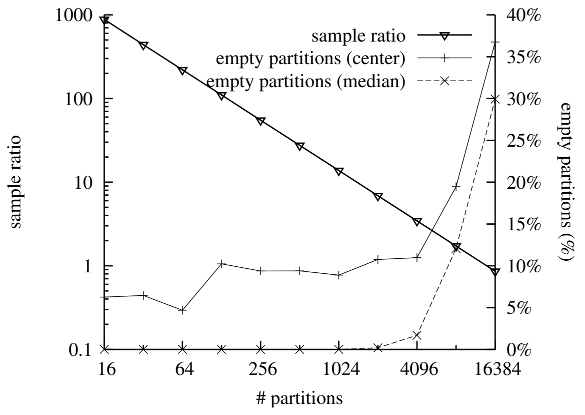
Using the sample ratio, we can explain the high percentage of empty partitions for small (0.01% and 0.1%) samples when creating large partitioning schemes. In Figure 3.7 we compared the four sample ratios (a) $r_{obs}^{0.01}$, (b) $r_{obs}^{0.1}$, (c) r_{obs}^{1}, and (d) r_{obs}^{10}, respectively. In order to better identify the correlation between sample size and the percentage of empty partitions, we also depicted the empty partitions of both quadtree variants.

In Figure 3.7(a), the first curve is $r_{obs}^{0.01}$, the sample ratio for the 0.01% training sample of P_{obs}. Increasing the number of partitions, we have about as many (or fewer) objects as partitions, i. e., the sample ratio (shown on the left-hand axis) decreases. Thus, with decreasing sample ratio, a partition is empty with increasing probability. The both other curves are the empty partitions of the quadtree variants, showing how the percentage of empty partitions raises. Therefore, our implementation did not construct partitioning schemes when a completely empty partition was split further.

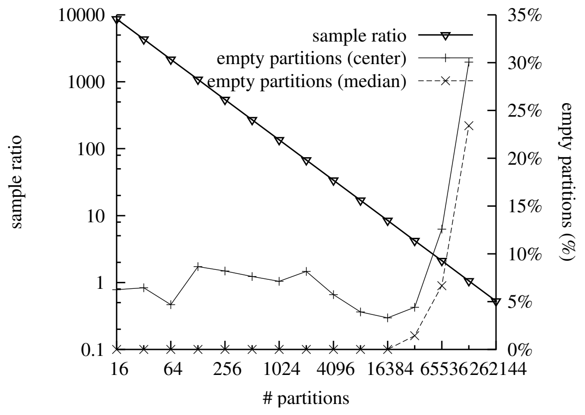
The observations for the 0.1% training sample (Figure 3.7(b)) are similar for both data sets. As soon as the sample ratio decreases below 10 (on the left-hand axis), the number of empty partitions increases significantly. Even for the 1% training sample (Figure 3.7(c)) this effect is visible when comparing the two last partitioning schemes evaluated with those from the 10% training sample (Figure 3.7(d)). Between the last partitioning schemes the sample ratio r_{obs}^{1} drops below 10 ($r_{obs}^{1}(131\,072) \approx 10.5$ and $r_{obs}^{1}(262\,144) \approx 5.3$) and there are more empty partitions. The ratio r_{obs}^{10} stays constantly above 10 and thus the number of empty partitions remains undistorted. To summarize, a training set is only suitable for the training phase, if its sample ratio is sufficiently high (> 10).

3.3.6 Baseline Comparison

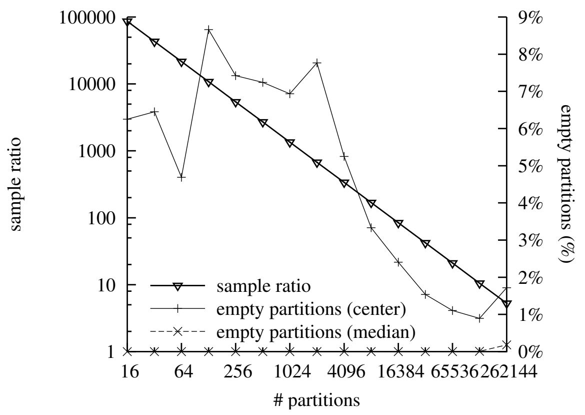
To evaluate the quality of partitioning schemes generated during the training phase, we compared the partitioning schemes obtained from our training samples with the partitioning scheme computed using the complete data as training sample, the baseline histogram. For illustration, we discuss the results for the quadtree-based partitioning schemes of P_{obs} with 1 024 partitions. The partitioning schemes determined on the basis of the different training samples (0.01%, 0.1%, 1%, and 10%) were almost identical to the scheme computed on the basis of the com-



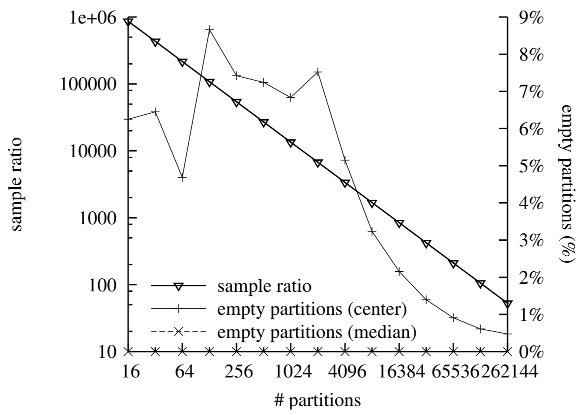
(a) 0.01% training sample



(b) 0.1% training sample



(c) 1% training sample



(d) 10% training sample

Figure 3.7: Effect of decreasing sample ratio, P_{obs}

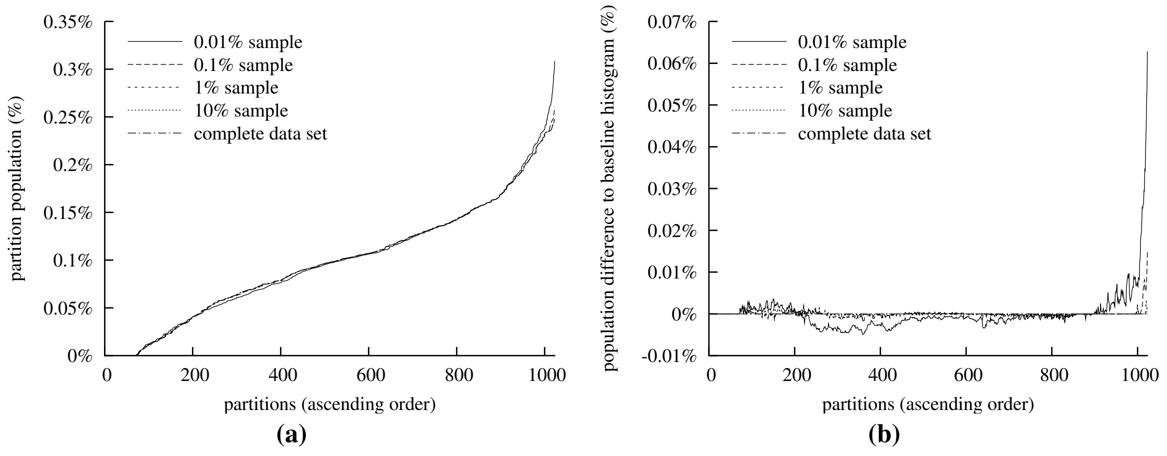


Figure 3.8: Baseline comparison for the standard quadtree, P_{obs}, and 1 024 partitions

plete data. Thus, in this particular case (quadtrees, P_{obs}, 1 024 regions), all our training sets yield very good partitioning schemes. Figure 3.8(a) shows the population of each individual partition relative to the size of the corresponding training sample, sorted in ascending order. We can see that the training samples are indeed representative because the distribution of the population is similar for all samples as well as for the original data. That is, the first partition contains no data and the last partition contains between 0.25% and 0.31% (for the 0.01% training sample) of the samples or the complete data set, respectively. In Figure 3.8(b), we see that the differences compared to the baseline histogram are negligible. The histogram created with the 0.01% training sample has the most differences, but the maximum deviation is at 0.07%. Note that computing partitioning schemes based on complete data sets is prohibitively expensive when dealing with very large current or future data sets, especially if verifying the median heuristics. Thus, a baseline comparison as the one above is infeasible under such circumstances.

3.3.7 Discussion

We shortly summarize the observations made during our evaluation. The training phase itself does not take very long, so it could be worthwhile comparing several training sets. Generating all 250 partitioning schemes[1] lasted about 120 minutes in our evaluation setup. This, for example, allows the detection of non-representative samples, if the curve in Figure 3.8 diverges too much from the curve of the complete data set. If a baseline comparison is infeasible, for example when using partitioning schemes such as median-based quadtrees, outliers can at least indicate differences between the drawn data samples. If training samples have a sufficiently high sample ratio, comparatively small data samples already provide good partitioning schemes. In our evaluation, using 10-times more data objects than partitions provided a good rule-of-thumb for reliable results. For our skewed data sample, median-based quadtrees achieved the best load-balancing and had quite homogeneous average data populations. Depending on the data structure, the number of partitions influences the data distribution characteristics. For example, 2-dimensional quadtrees should be generated with powers of 4 as partition cardinalities. Besides the parameters used in our evaluation, there are several other means to compare the data

[1]To avoid splitting empty partitions, we created only 11 quadtree-based histograms from the 0.01% training sample and only 14 histograms for the 0.1% sample, respectively. Thus, we created $2 \cdot (11+14+15+15) = 110$ histograms for each quadtree variant and $2 \cdot 15 = 30$ zone-based histograms.

structures and further properties of the data or queries that can be relevant for choosing the best partitioning scheme. If the partitioning scheme changes frequently and is transmitted regularly, the size of the data structure is also relevant for the decision process. Furthermore, considering a typical workload during the training phase can give further valuable insights as we will discuss in Chapter 7.

3.4 Related Work

Creating general purpose or application-driven multi-dimensional index structures has been and continues to be an active field of database research. Besides the survey of Gaede and Günther (1998), the book by Samet (2006) provides an encyclical source for index data structures. On the application-driven side, Csabai et al. (2007) compared the performance of k-d trees and Voronoi diagrams for scientific data sets. They also experienced boxes of multi-dimensional k-d trees to elongate along the second and higher dimensions, a similar effect as in Figure 3.1(b).

3.5 Summary

In this chapter, we describe a flexible framework for investigating community-specific index structures used as a partitioning scheme for data grid federations. Distributing the data across several grid resources to level skewed data distributions while preserving spatial locality yields improved throughput and better load balancing for data-intensive applications. We evaluated quadtree variants and the zones algorithm and their capabilities to partition data from several repositories for federated data grids. Our criteria can be applied or extended in many other e-science communities. The choice of mapping the data partitions to the grid nodes is up to the community. Possible options are random distribution or using space filling curves in order to further preserve data locality. The trade-offs between these various choices are discussed in the following chapter.

CHAPTER 4

Community Placement: Better Serving Locality with Space Filling Curves

Within this chapter, we discuss how the data partitions are mapped to the nodes within the community-driven data grid. We especially want to focus on *which data mapping strategy improves query processing throughput*.

We focus on two general options for achieving a data mapping to the nodes: *random* distribution and using a *space filling curve* (Section 4.1). After discussing the rationales for either choice, we evaluate the different placement strategies (Section 4.2). From our results, we conclude that using a space filling curve for data placement is preferable to a random assignment in many application scenarios.

4.1 Random or Space Filling Curves

While the previous section focused on balancing the data load on the partitions from our partitioning scheme, we discuss now the options for mapping those partitions to data nodes.

Within a central cluster environment, it is common practise to distribute the data partitions *randomly* on the individual nodes in order to achieve a good data load balancing. This is partially also motivated by the high-bandwidth interconnections which render the communication between the nodes a less important bottleneck as in a distributed scenario. Assuming a uniform workload with queries that have a high data locality, such random placement introduces a high parallelism. However, this comes at the cost of sacrificing the spatial locality between neighboring data areas. Changes in the access patterns, e. g., the researchers are interested in larger sky areas, might result in increased communication overhead.

Space filling curves aim at preserving this spatial locality to some extent while providing a linearization or a one-dimensional mapping for multi-dimensional data objects (as shown in Figure 2.3 on page 12 in our overview section). Closeness within the data space should result in a low distance within the identifier space. Space filling curves have been discussed extensively in the literature (e. g., Samet, 2006) and we therefore only consider the most prevalent, the Z-order (Orenstein and Merrett, 1984) and the Hilbert curve (Hilbert, 1891). Range queries in multi-dimensional space are translated to several interval-queries for the one-dimensional mapping. Asano et al. (1997) propose a space filling curve for squared queries in two-dimensional space which is optimal in the sense that it guarantees to need only three one-dimensional intervals to answer a query. Remarkably, new interesting insights about space filling curves and their locality properties have been investigated very recently by Haverkort and van Walderveen (2008).

Besides the fundamental aspect of preserving spatial locality, other factors can also influence the data mapping, such as the workload characteristics or the individual node capacities. For example, distributing popular data on multiple data nodes in order to increase the query parallelism is widely used in parallel databases and current data center layouts. In our opinion, preserving locality is the key in order to achieve a high throughput.

For the choice of the correct data placement, again properties of the partitioning scheme play an important role, e. g., the size of individual partitions. For community-driven data grids, small partitions enable us to migrate or copy partitions between nodes either for compensating a node failure or for replication purposes. For our quadtree-based partitioning schemes, the partitions are approximately of the same size by the *adoption to data distribution*. We therefore achieve data load balancing by placing roughly the same number of partitions on each node.

Within community-driven data grids, we prefer inter-query parallelism over intra-query parallelism and we therefore aim at sending queries to a single host where all data is locally available. We achieve this goal, if a query only requires one data region, as these regions are the most atomic data building block and guaranteed to be completely on a single node. When queries span multiple partitions and these partitions are all managed by a single node, no additional overhead is necessary in order to merge or transmit intermediate results. Otherwise a coordination node needs to be selected during query processing, as discussed in Section 6.1.

As described in the architectural overview (Section 2.2), our HiSbase prototype realizes the data mapping by using the key-based routing functionality of an overlay network. The region identifiers are uniformly mapped to the one-dimensional key ring. When a node arrives at the network for the first time, it receives a random identifier. During the join process, the arriving node is informed about its both neighbors and thereby can compute the range of identifiers, it is responsible for. The data is then transferred to this node by employing one of the dissemination techniques (either via data feeding or direct extraction from an archive) which are discussed in Section 5. Should a node return to the network, it reuses its previous identifier as it already has stored the relevant data locally.

4.2 Placement Evaluation

During our experiments, we evaluated how the various mapping options affect the data load balancing and query locality of our queries. We used the quadtree-based partitioning schemes created during the evaluation of the training phase presented in the previous chapter. Data load balancing and preservation of query locality are the two criteria to assess our various data placement strategies. A good data load balancing is achieved if all nodes manage the same amount of data. With regard to query locality, we evaluate how many nodes are required in order to answer a single query.

Queries that intersect a single partition are answered by a single node and thus have optimal query locality. We note that this is independent of the placement strategy because a partition is not further split across multiple nodes. However, queries spanning multiple partitions that are

Parameter	Value(s)	Description
P	P_{obs}, P_{mil}	Data set used for training sample extraction
Q	Q_{obs}, $Q_{mil'}$	Query workload used for query locality evaluation
s	0.1%, 1%, 10%	Size of the training set
m	Z-order, Hilbert curve, random	Mapping strategy
p	4^2, 4^3, 4^4, 4^5, 4^6, 4^7, 4^8, 4^9	Size of the partitioning scheme
n	10, 30, 100, 300, 1 000, 3 000, 10 000	Network size
r	1 000	Number of runs

Table 4.1: General parameters for the evaluation of mapping strategies

distributed to multiple nodes introduce the overhead described above (i. e., additional messages and intermediate results). With a good placement strategy, many of these (multi-region) queries can still be answered by a single node.

Compared to the evaluation setup of the training phase, we only used histogram sizes, which could be created directly from quadtrees (i. e., powers of 4 in our application scenario). The network size was changed between 10 and 10 000. As data sets we used both the observational data set P_{obs} and the simulation data set P_{mil}. We used our observational query set Q_{obs} and a synthetic workload $Q_{mil'}$ to test query locality. $Q_{mil'}$ consists of one million queries with a hot spot in the center area and search radii taken uniformly from 0.5 arc min, 1 arc min, 4 arc min, 1° and 4°. This set of radii was suggested by Nieto-Santisteban et al. (2007) as parameters for cone searches. By using the simulated data set with a workload that also contains larger queries, we can especially evaluate the effect on spatial locality.

For each mapping method–Z-order, Hilbert curve, and random placement—we performed 1 000 runs for each histogram size, data set, and network size combination. The figures show the average taken from these 1 000 runs and we report only on our findings from the 10% data sample. Furthermore, we consider only partitioning schemes whose size is larger than the number of nodes. With fewer partitions, the size of the individual partition increases and thus the transmission takes longer, e. g., if the partition needs to be transferred to another node. Table 4.1 summarizes the various evaluation parameters.

4.2.1 Data Load Balancing

We only discuss the data load balancing results of the observational data set P_{obs} as the results of the simulation data set showed a similar trend.

In the data load balancing part, we only discuss the result on the observational data set P_{obs}, as the trend in the results of the simulation data set P_{mil} was similar. Furthermore, the results for both space filling curves were the same with regards to data load balancing capabilities.

In Figure 4.1(a), we compare how the P_{obs} data load of 1 024 partitions is distributed across 1 000 nodes using a Lorenz curve (see Pitoura et al., 2006). On the x-axis we ordered the nodes according to the amount of data they are responsible for. For a system that optimally (uniformly) balances the data load, the Lorenz curve is close to the diagonal. For the shown configuration, we see that a high percentage (between 28% and 35%) of the nodes cover no data at all. Random

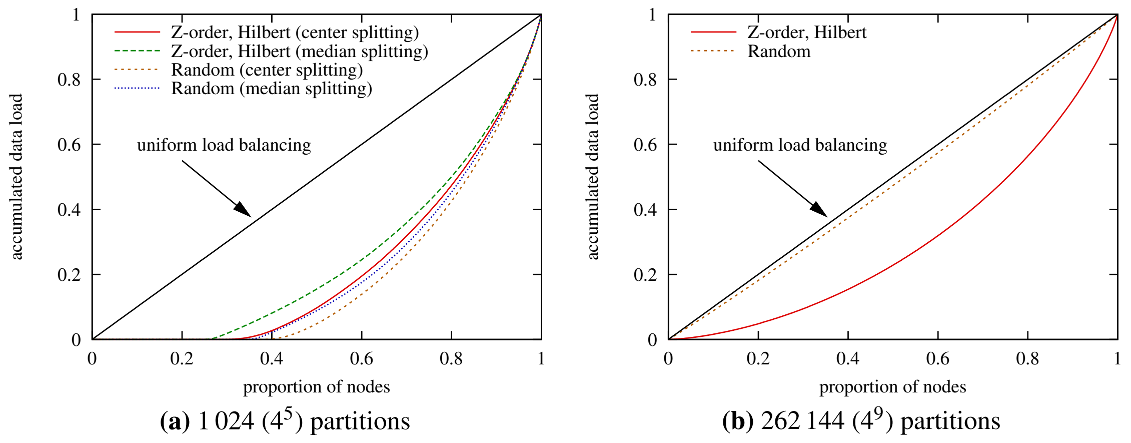


Figure 4.1: Data load balancing for the P_{obs} data set on 1 000 nodes

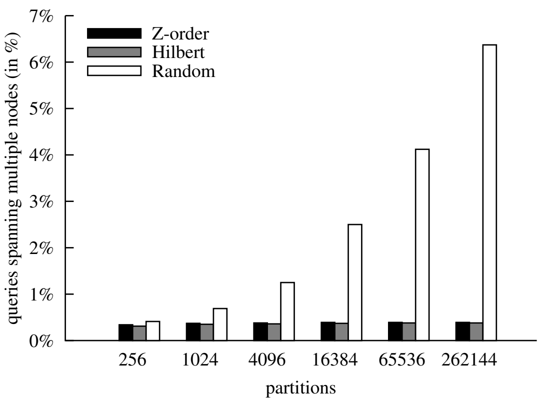
	Minimum	Maximum	Total
Z-order	162 MB	3.8 GB	44 GB
Random	828 MB	13 GB	129 GB
Space increase	511%	342%	293%

Table 4.2: Additional space required by placing partitions randomly on 32 nodes

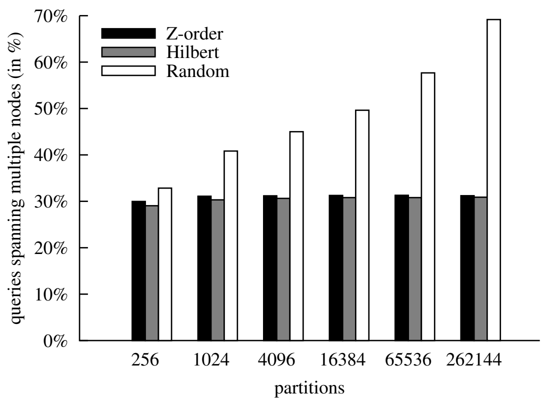
strategies are slightly worse than the strategies based on space filling curves. Due to the random placement of peers and the low region-to-node ratio, it is very likely that nodes do not cover any data region at all. We see a small benefit for the median based approach, as it adapts slightly better to the data load balancing.

In Figure 4.1(b), we show the distribution of 262 144 partitions to the same 1000 nodes. We see that the data load distribution improved for both space filling curves, as the curve is closer to the diagonal and less nodes cover no data. Placing the partitions randomly on the nodes is close to the optimal data distribution. We attribute this to skew that spans multiple partitions (neighbor partitions that have all little or all much data) which is potentially preserved by a space filling curve whereas random placement "shuffles" these regions.

With regards to completely partitioned data management, random placement clearly offers the best data load balancing. However, especially for our cross-matching scenario, random placement introduces a considerable storage overhead. For cross-matches, a small overlap at the borders of the partitions is necessary in order to find corresponding matches that are just "beyond the border". If a node covers many neighboring regions this overlap is not necessary for the "interior" borders, i. e., for borders between regions that are covered by the node itself. A concrete comparison for a network with 32 nodes is given in Table 4.2. By adding a small border of 0.1 degrees to each of the 262 144 partitions, the storage requirements triples for all node databases in total. The size of the smallest partition is even increased by a factor of five. Similar scaling-relationships were found when comparing the size of the CSV files for each node.

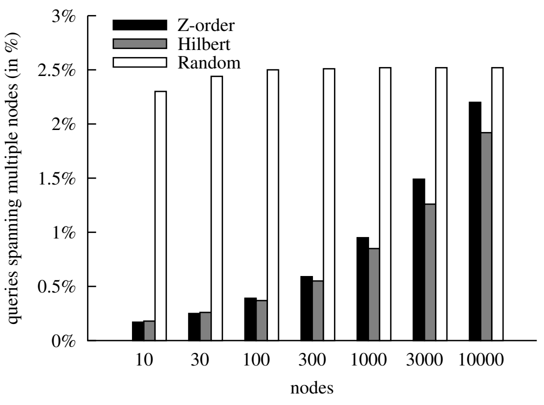


(a) Q_{obs}

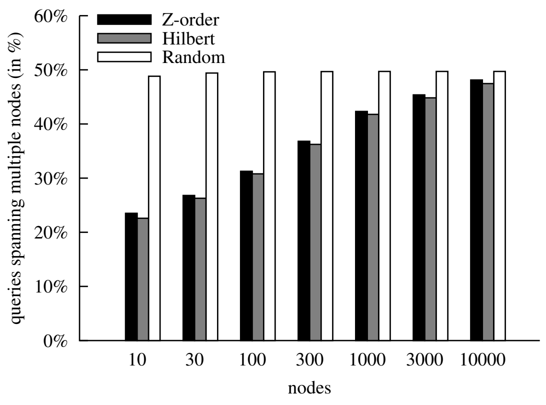


(b) $Q_{mil'}$

Figure 4.2: Query locality on 100 nodes with varying partitioning schemes



(a) Q_{obs}



(b) $Q_{mil'}$

Figure 4.3: Query locality for varying network sizes with 16 384 partitions

4.2.2 Query Locality

Besides data load balancing, we also evaluated the preservation of query locality for the mapping strategies. Figure 4.2 shows the percentage of queries that span multiple nodes during query processing while increasing the histogram size for a network with 100 nodes. For the observational workload Q_{obs} (Figure 4.2(a)), the queries exhibit a high degree of locality and when using space filling curves, only a tiny fraction needs data from several nodes. This query ratio increases for random placement when the partitioning scheme contains more partitions up to 6.4% percent. Though the queries span multiple regions when increasing the histogram size, the space filling curves ensure that the overall level of locality remains stable. Put differently: space filling curves are better in keeping the work on a single node.

This observation holds also for the simulation data set P_{mil} and query set $Q_{mil'}$, although on a different scale (Figure 4.2(b)). This workload has 30% queries spanning multiple nodes and both space filling curves are good in keeping the ratio stable. When used with 262 144 partitions, randomly placing the partitions on nodes results in 70% of the queries to span multiple nodes. The results for other network sizes show similar trends.

Above results suggest that it is reasonable to create a partitioning scheme with many partitions, even if the initial number of nodes is small. The locality-preservation by the space filling curves will still keep queries on a single node. Thereby, HiSbase can accommodate situations when institutions join with several nodes at once. As a side effect the size of the partitions becomes even smaller.

Figure 4.3 again shows the percentage of queries that require data from multiple nodes. In contrast to Figure 4.2, the histogram size is fixed at 16 384 partitions and shows the evolution when additional nodes join the network. The more nodes are added, the more both space filling curves approach the values of the random partitioning.

4.3 Summary and Future Work

Achieving good data load-balancing as well as preserving query locality is important for scalable data management solutions in e-science environments. Therefore the choices of partitioning scheme and of mapping strategy contribute significantly to how well the initial setup of the community grid will scale. From our experiments we draw two conclusions as advice for building community-driven data grids: 1) create a partitioning scheme with many partitions and 2) use space filling curves for mapping the partitions on the available nodes. Through this combination the query locality remains stable for a network with fixed size. When adding more nodes to a running setup the query locality for a fixed partitioning scheme remains higher than for placing the partitions randomly. In very restricted application scenarios, e. g., when queries are extremely local (one partition) and no “boundary data” is necessary, random placement offers a reasonable choice as the data load balancing capabilities are amenable to their full extent. Further studies which apply advanced space filling curves (e. g., Asano et al., 1997; Haverkort and van Walderveen, 2008) probably will provide additional insights.

CHAPTER 5

Feeding Community-Driven Data Grids

Scientific archives currently provide access to already existing huge data sets. Moreover, recently started scientific instruments, such as LOFAR and LHC, generate data at an enormous speed and scale. In order to cope with this data deluge, scalable high-throughput data management infrastructures, such as HiSbase, need to employ efficient data dissemination techniques.

Within this chapter, we discuss *how to efficiently distribute data within community-driven data grids*. First, we describe several scenarios where we apply data feeding to transmit data between the collaborating data nodes (Section 5.1). Then, we compare a *pull-based* approach, where each client pulls the required data, with two basic *push-based* techniques, where the archives publish the data to the clients either tuple-wise or the complete data at once (Section 5.2). We identify the major limitations of these approaches in the face of the requirements of e-science data grids and propose efficient *bulk feeding* techniques that transmit data in chunks. In order to tune our feeding strategies, we describe a model that optimizes data feeding by using paths with minimal latency and maximum bandwidth (Section 5.3). As we cannot use an optimal solution due to the problem's complexity, we present our *chunk-based* feeding techniques. We describe how to optimize the network traffic, the configuration of the feeding strategies, and the data processing at the receiving nodes in Section 5.4. Our evaluation results (Section 5.5) show that our bulk feeding techniques considerably accelerate data feeding. We conclude this chapter with related work (Section 5.6) and give an outlook on future issues (Section 5.7) in the context of data dissemination within scientific data grids.

5.1 Feeding Scenarios

In the following, we discuss five exemplary use cases for data feeding within community-driven data grids. They range from the initial feeding to node arrivals, node departures, and to data distribution according to an additional histogram. Nodes that receive feeding messages in our scenarios directly store the data in their local database.

5.1.1 Initial Load

The initial load scenario distributes data from an archive for the first time into the community-driven data grid. Although some archives maintain a few globally distributed mirrors in order to provide redundant access to the public data set, we only use one data copy during our initial feeding. For each catalog, one node contacts an archive and transfers the data to the network nodes.

5.1.2 New Node Arrival

Whenever a new node joins the network, it will get two direct neighbor nodes (one clock-wise, the other counter-clock-wise). By the underlying key-based routing fabric, these nodes are notified about the arrival of the new node. Both neighbor nodes determine the data subset, which is relevant for the new node, by the histogram and information extracted from the update message which is triggered by the arrival. The neighbors extract this subset and send it to the new node. Once the new node has received all its data, the node participates in query processing.

5.1.3 Planned Node Departure

The planned node departure is the symmetric event to the arrival of a new node. In contrast to both previous scenarios, now the databases at the receiving nodes already contain data. When a node leaves the network in a controlled manner, its data need to be transmitted to its neighboring nodes. Based on the identifiers of its both neighbors, the leaving node can identify the data which is added to the database of either node. Once it has finished the feeding process, the node leaves the network.

5.1.4 Unplanned Node Departure

In case of an unplanned node departure the former neighbor nodes are again notified using the underlying key-based routing system. At this point in time, both neighbors share the responsibility for the data regions once covered by the disappeared node. In cases where the outage of the node cannot be resolved within a predefined time frame the data needs to be retransmitted to the affected nodes. In this case, we extract from the archive only the regions that have been covered by the node that has left the network.

5.1.5 Replicating Data to Other Nodes

Once the data is distributed among the network nodes, their databases can serve as data sources for further data transfers. This is particularly useful when redistributing data for load balancing purposes. Chapter 8 describes the actual load balancing mechanisms in more detail. Here, we only discuss how to replicate all available data, e. g., in order to increase the data availability by an additional copy. Obviously, the redistribution is determined by a second, different histogram. Otherwise, all nodes would manage the same data twice and send the data to themselves.

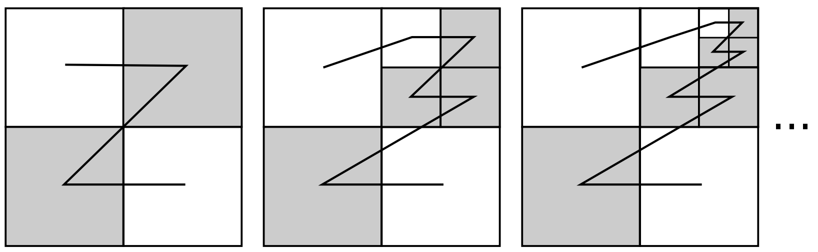


Figure 5.1: Example for geometric predicate optimization with only minor improvements

5.2 Pull-based and Push-based Feeding Strategies

For the following discussion, we summarize the expected figures for the number of data sets and their accumulated size, the number of nodes, and the anticipated number of data partitions within a community-driven data grids. We assume that our data grid will serve about 3–300 data sets whose accumulated size will comprise several terabytes up to petabytes. We anticipate that the networks will start with several nodes at the beginning and if new institutions join they will bring a considerable amount of nodes into the network. Thus, the network size is about ten times larger than the number of catalogs (30–3 000 nodes). In order to anticipate network growth, the number of partitions is a factor of hundred higher than the number of nodes. Thus, histograms have between thousands and hundreds of thousands of regions. Moreover, we get a reasonable size for the individual histogram region, e. g., for a petabyte data set the size of one region out of 300 000 regions would be in the gigabytes. To sum up, we expect to have far more regions than nodes and more nodes than individual catalogs.

5.2.1 Pull-based Feeding

For delivering the data from the catalogs stored at the original data sources (archives) to the individual nodes within a community-driven data grid, nodes could *pull* their data to their local database. This approach follows a traditional client-server model: the nodes are the clients and the archives act as servers. When a node requests its data sets, it first generates the SQL query for each catalog from the dimensions of its covered regions. It then submits the query to the database on the archive server, retrieves the results, and stores the data in its local database. Pull-based feeding has the advantage that it is easy to implement and all nodes directly communicate with the archive servers. *Direct communication* denotes, that receiver and sender are connected directly using a TCP/IP connection, and not necessarily a direct physical wire. On the downside, the approach burdens a high load on the nodes serving the data sets. In the initial load scenario, for example, all nodes request the data at once and in parallel. Furthermore, if nodes cover many regions query predicates potentially get too complex for the query optimizer of the database system. For example, a node within a 100-node network using a histogram with 300 000 regions would cover about 3 000 regions. For our running astrophysics example, this would result in an SQL query with 6 000 range predicates (one predicate for the right ascension attribute and one for the declination attribute for each region). Simplifying predicates by geometric optimizations is in general applicable for quadtree-based histograms. For example, the four predicates of adjoining square regions could be combined to two predicates (defining a rectangle). In the worst case, as shown in Figure 5.1, this technique can yield only minor simplifications. Due to these limitations of the pull-based approach, we now discuss push-based dissemination techniques.

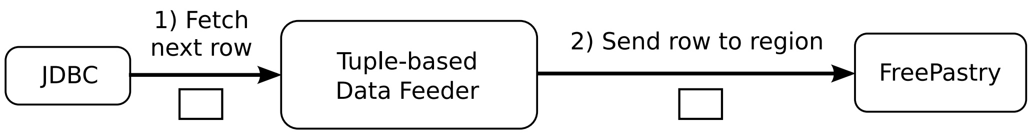


Figure 5.2: Tuple-based feeding strategy (TBFS)

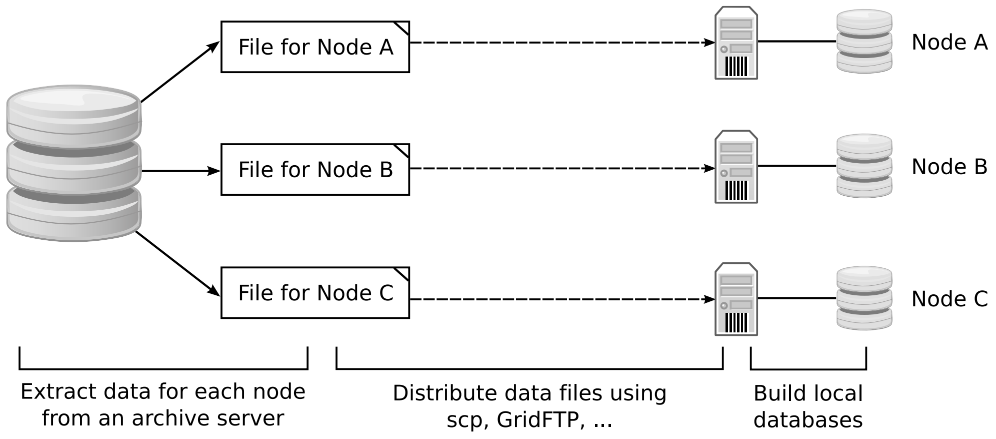

File for Node A
File for Node B
File for Node C
Node A
Node B
Node C
Extract data for each node from an archive server
Distribute data files using scp, GridFTP, ...
Build local databases

Figure 5.3: "Wolf"-based feeding strategy (WBFS)

5.2.2 Push-based Feeding

For push-based feeding strategies, the *feeder*, i. e., the sending node, actively publishes its data to the network. All push-based strategies have in common that they determine receivers with our histogram and hence considerably simplify the database queries at the feeding node. For example, during the initial load, the feeder simply performs a sequential scan on the complete catalog in order to retrieve the relevant data.

Tuple-based Feeding Strategy (TBFS)

For the tuple-based feeding strategy, the feeding node feeds each individual tuple into the network after it has determined the region for the tuple. This basic transmission process is depicted in Figure 5.2. If partitions have an overlap at the border to their neighbors, tuples falling into that border are replicated to both nodes.

Related P2P information systems (e. g., Crainiceanu et al., 2007; Ganesan et al., 2004a), which primarily scale with network size and are designed for a highly volatile network, use this data dissemination technique. While the TBFS offers good flexibility for low data volumes, this approach does not scale for the anticipated data set sizes as it introduces too many messages—one message for each tuple.

The messages are small and directly sent via the P2P overlay network. However, data is extracted fast from the archive and therefore we need to fine-tune the message handling. If the feeding speed is not configured correctly, the outgoing queues at the feeder will be swamped and either will get very large or messages will be discarded. To devise a good *deceleration* of the feeding nodes, i. e., fixing the frequency of outgoing messages, is difficult and prolongs the feeding process further.

“Wolf”-based Feeding Strategy (WBFS)

The “wolf”-based feeding strategy (WBFS) distributes the data for each node in a single big message and is depicted in Figure 5.3.[1] Assuming that the positions of the nodes are known in advance, this strategy uses a single sequential scan on each catalog in order to extract the individual tuples. Similar to the TBFS, this strategy identifies tuples which need to be sent to multiple nodes. Instead of creating a message for each such tuple, this strategy stores all tuples for a specific node in appropriate data files (e. g., CSV files). These files are then transmitted to the data nodes using a suitable protocol such as *scp*, *rsync*, or *GridFTP*. Each node compiles its own database based on these data files.

Using the WBFS, we increase the parallelism of the data extraction at the sources and of the database creation at the receivers. Additionally, feeders and receivers use direct communication. However, the WBFS still has some major drawbacks. Although the WBFS is push-based, it requires a high storage capacity at the feeders in order to create the data files for all nodes in parallel. Furthermore, this strategy performs data extraction as a blocking operator. Only if all files for a particular catalog are completed, the data files are transferred to the appropriate nodes.

We see from the two basic push-based strategies, TBFS and WBFS, that is critical to define an appropriate size for data chunks, i. e., to define how many objects are combined into a single message. Using only one tuple per messages induces too many messages and sending all data at once offers only a limited parallelism and pipelining. We need to trade off high parallelism against network overhead when designing our *bulk feeding* techniques. For this purpose, we analyze how to optimize these feeding strategies by a mathematical model. Although the optimal solution is not applicable in our context, we derive several useful indications that have influenced our feeding strategies.

5.3 An Optimization Model for Feeding

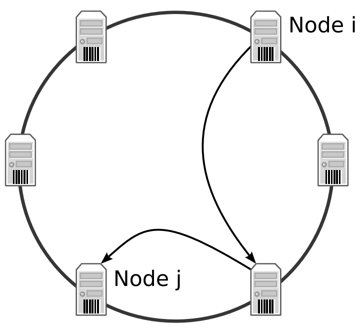
In order to optimize the feeding process, efficient data transfers are required to reduce both the load and space requirements on the archive nodes as well as the messaging overhead during transmission. The efficiency and speed of our feeding strategies is therefore determined by the latency and bandwidth constraints within the network. For creating a model which allows us to describe the latency and bandwidth constraints, we require a network snapshot of the current network configuration.

5.3.1 Network Snapshots

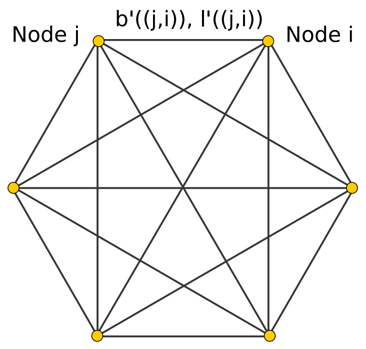
In oder to find the appropriate abstraction of the network configuration, we compare different views of the communication network of community-driven data grids as shown in Figure 5.4.

In general, we use the key-based routing protocols of the overlay network (Figure 5.4(a)). However, we cannot directly derive the properties of the physical network—such as latency, bandwidth, or closeness between nodes—from the overlay network links. Overlay links abstract on purpose from the actual network and this fact renders this view of the network as infeasible for building a mathematical model for latency and bandwidth constraints.

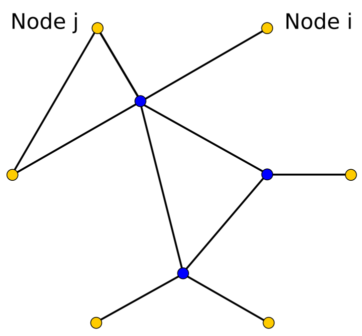
[1] Due to the fact that nodes get a single “big bite”, we denoted this strategy “wolf”-based inspired by the verb *to wolf: to eat greedily* (source: http://www.merriam-webster.com/dictionary/wolf).



(a) Network snapshot using overlay paths



(b) Physical network snapshot as complete undirected graph $G' = (V', E', l', b')$



(c) Physical network snapshot as undirected graph $G = (V, E, l, b)$ with transit nodes (inner nodes)

Figure 5.4: Overview of network snapshots

In the following, we will use specific *network snapshots* that enable us to better describe the physical network configuration:

Definition 5.1 (Network Snapshot) A network snapshot is a graph $G = (V, E, l, b)$ that describes a static view of a physical network. Let V be the set of nodes within the physical network and E the set of communication links (i, j) between two nodes $i, j \in V$. Then let function $l : E \rightarrow \mathbb{R}^+$ define the latency of an edge and function $b : E \rightarrow \mathbb{R}^+$ define the bandwidth of an edge, respectively. □

One example of such view of the physical network is the complete undirected graph, as shown in Figure 5.4(b). This model represents the direct communication paths between all nodes participating in the common data grid. The graph is complete, as we can assume that all nodes within a community data grid allow for direct communication, i. e., all nodes within the network can establish pairwise connections. Furthermore, the graph is undirected, as data grids usually use bidirectional, high-bandwidth communication links. Unfortunately, this view does not capture cross-traffic, i. e., traffic between two independent pairs of nodes that will influence each other's performance as they share a common subpath in the physical network.

We will therefore use network snapshots that include *transit nodes* and that enable us to create a more realistic network model. Transit nodes represent those routing nodes that connect several nodes within the same institution or campus to the other nodes within the data grid infrastructure. If two nodes share a common transit node, they compete for the transit node's bandwidth and influence each other. For example in Figure 5.4(c), node i and node j share a common transit node.

In order to create network snapshots, one needs to measure connection channels between nodes. In order to collect latency and bandwidth information network, nodes could ping other nodes or transmit sample data, respectively. Transit nodes on the communication paths could be identified using tools such as *traceroute*. Caching the information of previous transfers could also be a suitable mean to collect such network snapshots. We refrained from modelling further aspects such as failing nodes or skewed demand on the individual links as these would further add to the complexity of the problem.

Thus, we use network snapshots with transit nodes for our model in order to identify optimal paths for data dissemination. We consider a path optimal, if it offers both low latency and high bandwidth.

5.3.2 A Model for Minimum Latency Paths

First, we construct a model based on a network snapshot $G = (V, E, l)$ that only uses the latency function. During data feeding, messages should be sent on paths that offer the lowest latency possible. We show one approach for finding minimum latency paths based on Algorithm 5.1 and our network snapshot. The algorithm is a modified version of the *Dijkstra-Algorithm* (Cormen et al., 2001). Within the algorithm, we denote the set of nodes adjacent to a node subset $S \subseteq V$ as the *neighborhood* of S in network snapshot G.

Definition 5.2 (Neighborhood $N(G,S)$) For any node set $S \subseteq V$, we define the neighborhood of set S in network snapshot G as $N(G,S) = \{ n \in V \mid \exists\, s \in S : (s,n) \in E \}$. □

Algorithm 5.1: Minimum latency path

```
Data: Network snapshot $G = (V,E,l)$, latency function $l : E \to \mathbb{R}^+$, node $s \in V$
Result: Latency $L(v)$ of a minimum latency path for all nodes $v \in V$ reachable from $s$
begin
    $S \leftarrow \{s\}$
    $L(s) \leftarrow 0$
    $L(v) \leftarrow \infty\ \forall v \in V \setminus \{s\}$
    for $v \in N(G,S)$ do
        $L(v) \leftarrow l((s,v))$
    end for
    while $N(G,S) \setminus S \neq \emptyset$ do
        Choose $v_{min} \in N(G,S) \setminus S$ with $L(v_{min}) = \min\{ L(v) : v \in N(G,S) \setminus S \}$
        for $v \in N(G,\{v_{min}\}) \setminus S$ do
            $L \leftarrow \min\{ L(v),\ \max\{ L(v_{min}),\ l((v_{min},v)) \} \}$
        end for
        $S \leftarrow S \cup \{v_{min}\}$
    end while
end
```

Theorem (Minimum Latency Path) *Let $G = (V,E,l)$ be an undirected graph with the latency function $l : E \to \mathbb{R}^+$ and $s \in V$ be an arbitrary source. Algorithm 5.1 solves the problem of minimum latency paths by annotating nodes $v \in V$ with the latency $L(v)$ of a minimum latency path from s to v. Using a* depth-first search *that purges edges e with $l(e) > L(v)$ starting in s determines the actual minimum latency path.*

PROOF: Let S_i be the set S in step i. We show that in every step of the algorithm, $L(v)$with $v \in V$ is the latency of a minimum latency path from s to v that only uses intermediate nodes from S_i. In proof of the theorem we use induction over $|S| = i + 1$:

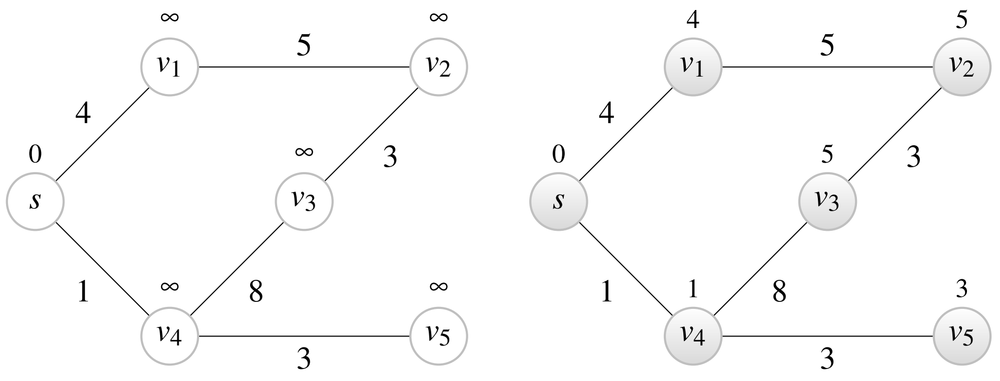
Base Case: For $|S| = 1$, set S_0 only contains node s and all nodes v in the neighborhood of s (i. e., $v \in N(G,s)$) are initialized with the latency $L(v) = l((s,v))$ of the direct edge to s (line 6 of Algorithm 5.1). All remaining nodes—those which are not in the neighborhood of s—are not reachable, because s is the only possible intermediate node ($S_0 = \{s\}$) at this point. Hence we label these nodes with ∞. Therefore, the current minimum latency paths, i. e., the direct edges, from the source s are correctly identified.

Inductive Step: Let the hypothesis hold for set $S_{n-1} = \{s, v_1, v_2, \dots, v_{n-1}\}$ for the inductive step. Therefore $L(v)$, with node $v \in V$, denotes the latency of a minimum latency path from s to v with intermediate nodes in S_{n-1}. The next step adds v_n to the set S_{n-1}. For all nodes $v \in S_{n-1}$, we have $L(v_n) \geq L(v)$, because $L(v_n)$ is the minimal latency annotation of nodes not in S_{n-1} and latency annotations are monotonically increasing. If now there exists a node $v^* \notin S_n = \{s, v_1, v_2, \dots, v_n\}$ for which exists a path using only intermediate nodes in S_n that yields a lower latency annotation than $L(v^*)$, this node must be a neighbor of node v_n. The hypothesis states that the labels $L(v_1), L(v_2), \dots, L(v_{n-1})$ do not change in step n. In step n, therefore, a path from s to v^* that uses only nodes in S_{n-1} as intermediate nodes cannot lead to a lower value for $L(v^*)$ than the one that is set after step $n-1$. Thus, the minimum latency path from source s to node v^* with intermediate nodes in S_n is made up of a minimum latency path from s to v_n with intermediate nodes only in S_{n-1} and latency $l((v_n, v^*))$. According to the base case, the latency $L(v^*)$ is $\max\{\, L(v_n),\ l((v_n, v^*))\, \}$. This is exactly the way label updates are handled in line 11 of Algorithm 5.1. Hence the hypothesis is justified.

For a proof of correctness for the pruning *depth-first search* algorithm to determine the actual minimum latency path we refer the reader to the respective literature (Cormen et al., 2001). ■

Running Time *Algorithm 5.1 runs the outer while-loop $|V|$ times as in each round a node is added to set S ($|S|$ is increased by 1) and set S initially only contains node s. In the worst case, the inner for-loop cycles $|V|$ times for a complete graph. Therefore the running time of Algorithm 5.1 is in $O\left(|V|^2\right)$. A* depth-first search *has a worst case running time of $O(|V| + |E|)$. Hence, the running time to find minimum latency paths is in $O\left(|V|^2\right)$.*

Example 5.1 We execute Algorithm 5.1 on the example graph G^* depicted in Figure 5.5(a) using s as a source node. For the sake of simplicity, edges are labeled with integer values instead of real latency values. Appendix A provides a stepwise execution of the algorithm. The resulting graph with latency annotations is depicted in Figure 5.5(b).



(a) Undirected graph G^*

(b) Result of Algorithm 5.1 for s as source node

Figure 5.5: Result of Algorithm 5.1 for G^* and s as source node

We assume that sending individual messages has only a negligible influence on the latency of the link. This allows us to reuse a network snapshot for all messages. We therefore create the network snapshot by running Algorithm 5.1 once for each node. Every time a message is ready for sending and we need to determine a minimum latency path, we only perform a *depth-first search* in the annotated network snapshot. We briefly sketch how to optimize our algorithm further by replacing the depth-first search with simple lookups. The basic idea of this optimization is to annotate nodes with their predecessor on a minimum latency path with regard to the specific source during the execution of Algorithm 5.1. For retrieving the path, we only need to trace the path from the target back to the source.

5.3.3 A Model for Maximum Bandwidth Paths

While the previous model only considered the latencies of the network, we now define a model which focuses on finding maximum bandwidth paths from a given network snapshot $G = (V, E, b)$. This problem is more complex than finding minimum latency paths because sending a message on a channel *does* influence the available bandwidth for other messages. We choose *flow networks* in order to model the information flow during the data feeding in our data grid infrastructure. To be more precise, we will consider our scenario as instance of the *multicommodity flow problem*. The multicommodity flow problem describes multiple different commodities that are transferred from their sources to their sinks through a common network (Awerbuch and Leighton, 1993). Thereby, our flow network G has the following properties:

Vertices $V = N \cup W$: The vertices $v \in V$ in our model are either data grid nodes $n \in N$ or *transit nodes* $w \in W$. We assume *transit nodes* do not participate in the data grid infrastructure, i. e., they only forward intermediate traffic.

Edges E: Edges are communication channels that can be used to transfer messages used during our data feeding. We assume that every network connection is bidirectional and offers equal bandwidth for both directions as assumed for e-science data grids in Section 5.3.1.

Bandwidth b: Each edge $(u, v) \in E$, with nodes $u, v \in V$ is assigned a non-negative bandwidth $b(\,(u, v)\,) \geq 0$. All edges that do not exist $(u', v') \notin E$ have bandwidth $b(\,(u', v')\,) = 0$. We assume that each node is also connected with itself and no bandwidth restrictions exist in this case: $\forall v \in V \; \exists (v, v) \in E : \; b(\,(v, v)\,) = \infty$.

Commodities $C = \{c_1, c_2, \ldots, c_k\}$: For modelling data feeding processes within community-driven data grids, commodities represent the individual messages used to transfer objects from the different catalogs (e. g., ROSAT, SDSS, or TWOMASS). Each commodity is specified by the triple $c_i = (s_i, t_i, d_i)$, $1 \leq i \leq k$, defining the source s_i and sink t_i of the packet, and its demanded bandwidth flow value d_i. How many commodities are used depends on how many objects are grouped together.

Sources $S = \{s_1, s_2, \ldots, s_l\} \subseteq N$: Considering data sources $s \in S$, each node can be a potential source (feeder) of data. The exact number of sources depends on the specific feeding scenario discussed in Section 5.1. For example, we have as many sources as catalogs in the initial load scenario whereas all nodes are sources in the replication scenario.

Sinks $T = \{t_1, t_2, \ldots, t_m\} \subseteq N$: Just as for sources, every node can be a sink (receiver) for messages. We note that sources can simultaneously be sinks for either commodities they induce to the network or messages from other feeding nodes.

Connectivity of G: We further assume that the graph G is connected. This property states, that for all nodes $v \in V$, all sources $s \in S$, and all sinks $t \in T$ a path $s \rightsquigarrow v \rightsquigarrow t$ exists.

Definition 5.3 (Bandwidth Flow Functions) Using our *flow network G* we can now define the *bandwidth flow functions* for commodities $c_i \in C,\ 1 \leq i \leq k:\ f_i : V \times V \rightarrow \mathbb{R}$ as follows:

Bandwidth capacity: $\forall u,v \in V : \sum_{i=1}^{k} f_i(u,v) \leq b((u,v))$. This states that all flows must not surmount the available bandwidth of the network links.

Skew symmetry: $\forall u,v \in V :\ f_i(u,v) = -f_i(v,u)$

Flow conservation: No node should retain a commodity if it is not its sink. Thereby, the in-flow must be equal to the out-flow on such nodes: $\sum_{u \in V} f_i(v,u) = 0$, for $u \in V$ that are neither a source nor a sink for commodity c_i. In particular this constraint must hold for all *transit nodes* $w \in W$ because they do not participate in the data grid and therefore cannot act as a sink.

Demand satisfaction: To satisfy the specified demand for all commodities, the source must have an accumulated outflow in size of the bandwidth in demand: $\forall c_i = (s_i, t_i, d_i)$: $\sum_{v \in V} f_i(s_i, v) = d_i,\ 1 \leq i \leq k.$ □

Our goal is to maximize the aggregated *bandwidth flow* in our *flow network*. Hence we maximize the demand for all commodities, so that each can be satisfied and the aggregated flow is maximized:

$$max \sum_{i=1}^{k} d_i = max \sum_{i=1}^{k} \sum_{v \in V} f_i(s_i, v),\ \text{with } s_i \text{ being the source for commodity } c_i \in C \tag{5.1}$$

When all nodes are centrally coordinated such that only a single feeding message is sent through the network, finding a *maximum bandwidth flow* is straight-forward. Messages correspond to commodities in our model, and therefore this case is a one-commodity flow problem and we can compute an optimal solution, e. g., by applying the *Ford-Fulkerson Algorithm* (Cormen et al., 2001). However, performing feeding according to this scenario would deteriorate the overall performance significantly.

According to the literature, our model for maximizing the bandwidth flow corresponds to the *undirected maximum multicommodity real flow problem.* Finding a polynomial-time algorithm for more than two commodities causes major difficulties (Even et al., 1975). The only known efficient way to compute the exact *maximum bandwidth flow* for our scenario according to Equation (5.1) is to solve a linear program because there are likely more than two commodities. Even though we can solve linear programs with polynomial-time algorithms (Cormen et al., 2001), the solution needs to be recomputed every time a message was transferred successfully or is about to be sent. Furthermore, all nodes in the optimal flow plan are involved in the calculation of routing decisions and usable bandwidth.

Awerbuch and Leighton (1993) describe a local-control approximation for *multicommodity flow*. The introduced algorithm is faster than solving a linear program. As it is a local-control approach, routing decisions do not have to be delegated to other nodes. The algorithm's running time is bound by $O\Big(|E|^3|C|^{5/2}L\varepsilon^{-3}\log|C|\Big)$ where L is a parameter that bounds the maximum path length of any flow from source to sink and ε^{-1} defines the accuracy of the approximated result.

5.3.4 Combining Latency and Bandwidth

Both models for the individual targets of minimizing the latency and maximizing the bandwidth show that quite an effort is necessary in order to provide an optimal solution. Solving both problems at once requires to solve two objective functions. However, solving the combined problem is even harder, as both parameters are not necessarily correlated. High bandwidth communication links not always have a low latency, such as satellite links. Thus, meticulous tuning is required in order to achieve an optimal solution.

5.3.5 Conclusions

Based on these findings, we aim for using our chunk-based techniques, which we describe in the following, instead of striving for an optimal solution. Nonetheless, we can draw several conclusions from the discussion so far.

Enabling Direct Communication to Nodes

Using the overlay routing fabric for data transfers during feeding incurs a high messaging overhead and probably results in sub-optimal dissemination paths. This implies that feeding nodes are required to generate a region-to-node mapping before they start feeding. Therefore, we need to communicate the mapping information efficiently between the nodes of our data grid. Once the mapping information is complete, the feeders can create data chunks according to nodes. Moreover, building the chunks based on nodes and not based on regions simplifies duplicate eliminations at the end of the feeding process. If the regions of a node are known, the feeder can ensure that objects that lie within the border of two regions of the same node are not sent twice.

Enable Parallelism and Pipelining during Feeding

Currently each project stores its data set in a separate database. Thus, the individual catalogs are prevalently stored at different sites. Therefore, we can use one node per catalog to disseminate the data into the network and use the available bandwidth of that node optimally for a single catalog. As these catalogs can be processed independently, the data distribution time is dominated by the distribution time of the largest catalog. If one node feeds several catalogs, e. g., during the replication scenario, it is reasonable to feed catalog by catalog as this favors to use the complete bandwidth for a single catalog. Once a node has finished its first catalog it can immediately begin to publish the next catalog.

Choosing a good chunk size improves the pipelining within the feeding process. Once a chunk has the configured size it can be forwarded to the receiving node for processing while the feeder already prepares the next chunk (potentially for a different node). We do not further consider other thresholds at the feeder side such as time thresholds, as these can be expressed by a size limit. Given a time threshold, we can compute the message size which is generated by the feeder in the given time and use that size as limit.

Flow Control at the Feeder

Finally, we implement flow control at the feeding nodes, as these can directly influence the amount of data within the network. Other strategies, such as buffering data at intermediate nodes, would ultimately need to stop the inflow at feeders requiring sophisticated congestion

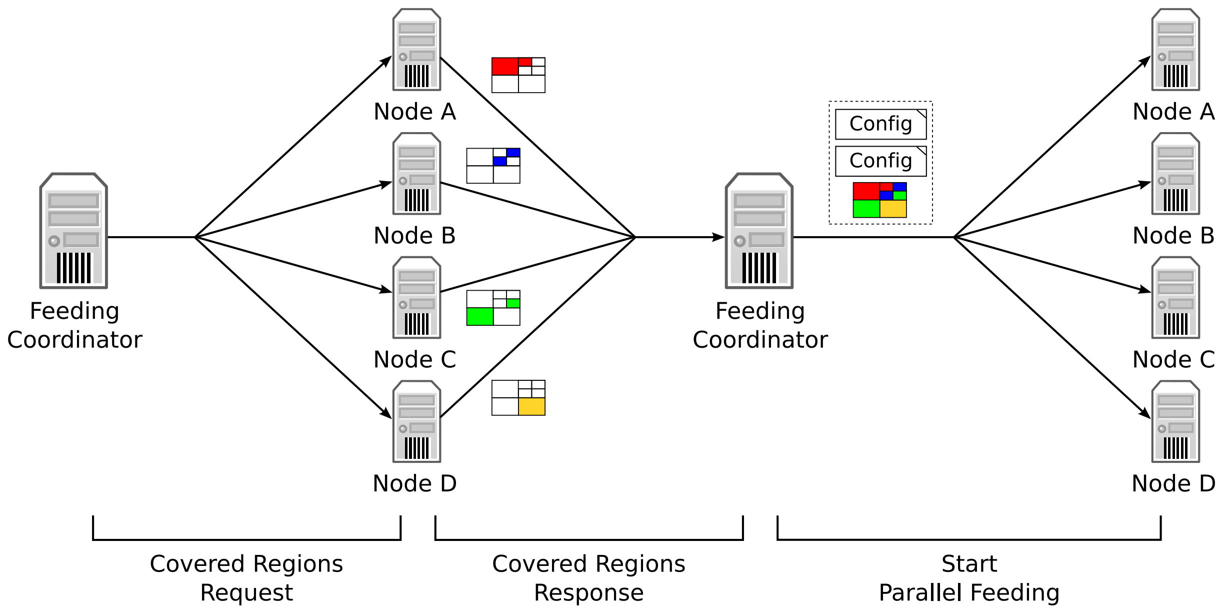


Figure 5.6: Communication pattern for creation of region-to-node mapping during data replication

control protocols. The aggregated network flow depending on the outflow at the sources in Equation (5.1) states that we should optimize the outflow at the feeding nodes as good as possible. Moreover, many feeding processes are likely to run in the background (except for the initial load). Therefore, we limit the number of parallel connections between two nodes as well as all outgoing parallel connections of a single node. Once one of these thresholds is reached the according feeder pauses feeding until the load is reduced.

5.4 Optimization by Bulk Feeding

In the following, we describe our different design choices with regards to optimizing the network traffic, the different chunk-based strategies, and the process of importing data at the receiver side.

5.4.1 Traffic Optimizations

One conclusion from Section 5.3.5 was to enable direct communication during the feeding process. For this purpose, a feeding node caches the region-to-node mapping, the so-called *CoveredRegionMap* locally. In order to exchange the mapping information at the beginning of a feeding process, the nodes communicate via a multicast-channel that transmits broadcasts efficiently.[1] The receivers of the broadcast return their covered interval of the key ring as acknowledgement. Thus the initiator of the broadcast can verify that all nodes have received its message. Based on these acknowledgement messages, the feeder builds its *CoveredRegionMap*.

We can further improve the exchange of mapping information during the replication process. All nodes will work as feeders and receivers, concurrently. In order to avoid the overhead for each node requiring the *CoveredRegionMap*, we identify a single node as *feeding coordinator*. This feeding coordination node initiates a data replication and realizes a *Two Phase-Commit*

[1]Scribe (Castro et al., 2002) is one such multicast-implementation based on the Pastry system.

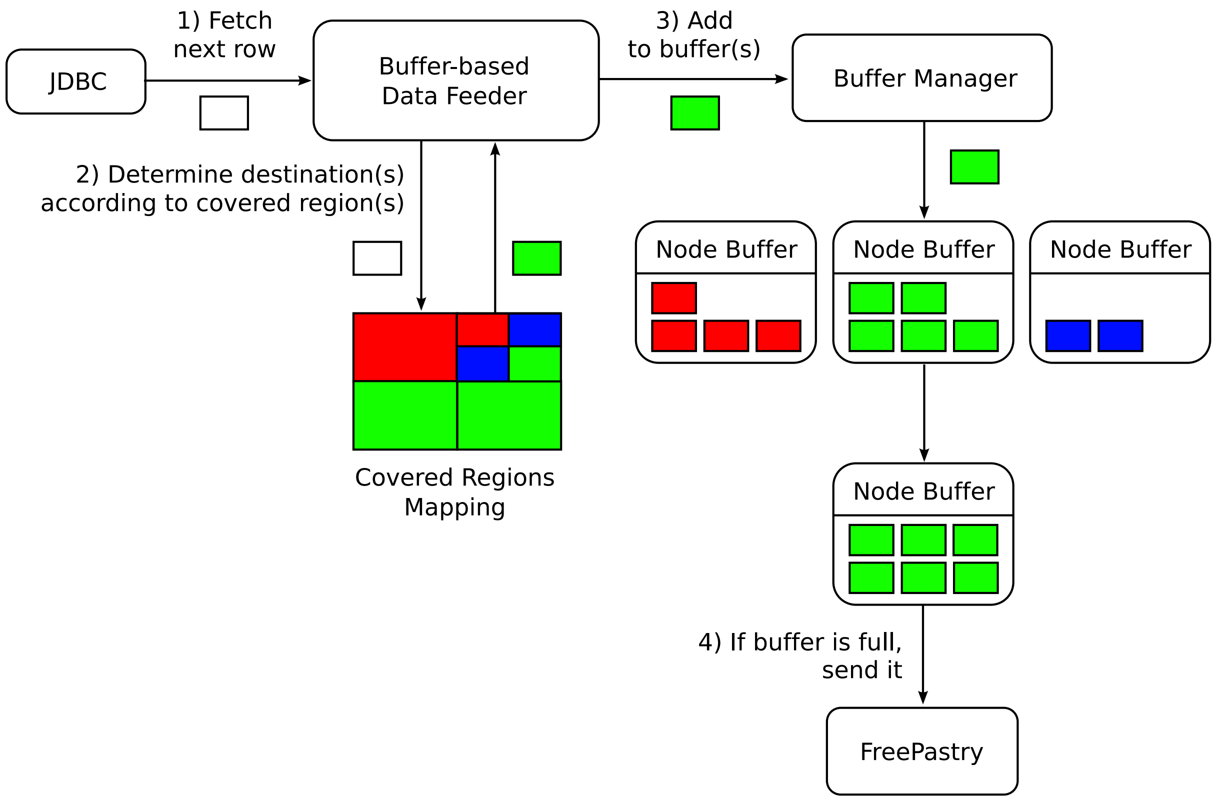


Figure 5.7: Buffer-based feeding strategy (BBFS)

(2PC)-like communication pattern as described in Figure 5.6. The feeding coordinator builds the *CoveredRegionMap* from the coverage information of all remaining nodes. The created *CoveredRegionMap* and the feeding configurations (e. g., what catalogs to feed) are distributed to all nodes. Once the other feeders receive the request, they can start their local feeding process and receive the messages from other nodes accordingly.

Besides enabling direct communication with the *CoveredRegionMap*, we tune TCP buffers and using several parallel streams which have been identified to be important factors for improving dissemination throughput (Yildirim et al., 2008).

5.4.2 Chunk-based Feeding Strategies

For our bulk feeding, we propose two chunk-based feeding strategies in more detail: a buffer-based and a file-based feeding strategy. The two strategies differ in the way how data objects are combined and transmitted between feeders and receivers.

Buffer-based Feeding Strategy (BBFS)

For the buffer-based feeding strategy, we extract a tuple from the archive and examine histogram and *CoveredRegionMap* to which node the tuple should be transmitted. If tuples are in the border areas of regions on different nodes, tuples are added to the buffers of all nodes affected. Once a buffer reaches the size limit it is transmitted to the according receiver. This process is depicted in Figure 5.7. The *BufferManager* ensures that if the number of parallel connections in general or to a particular node have been reached, the feeding is paused. Once it is notified that resources have been freed again, it resumes feeding.

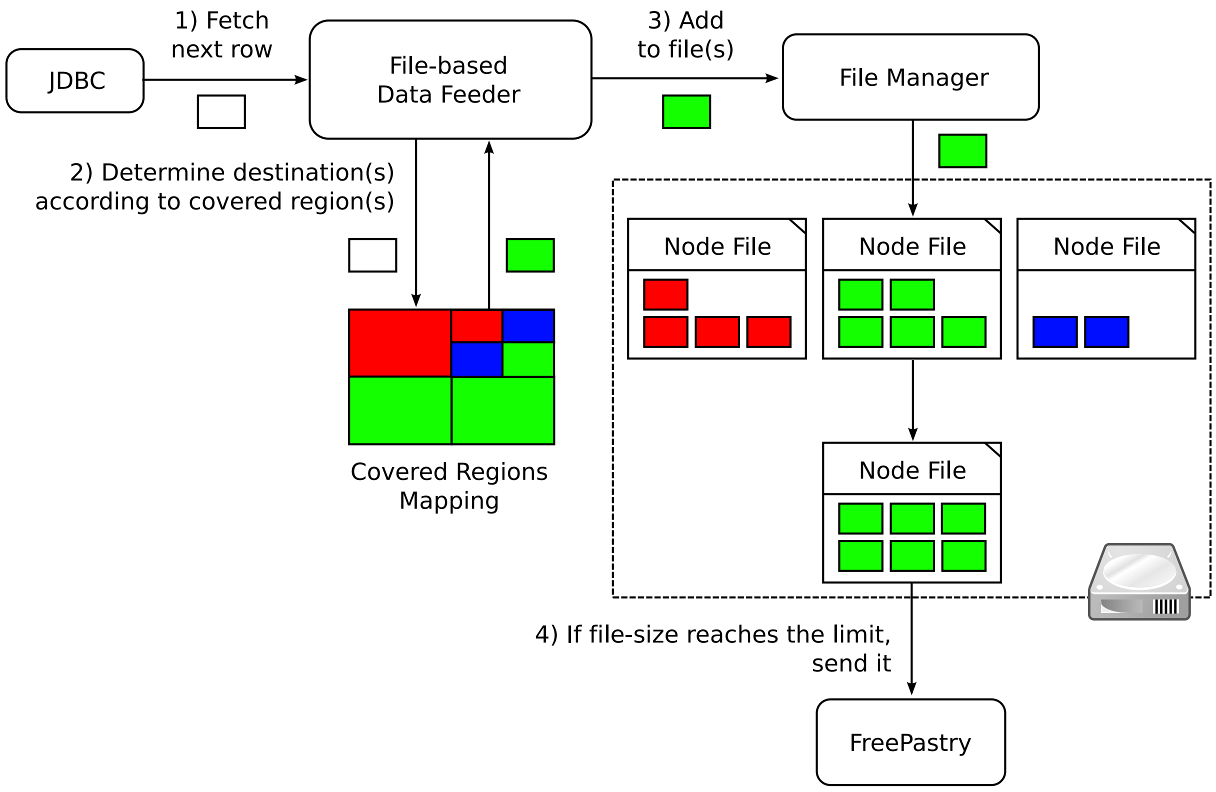


Figure 5.8: File-based feeding strategy (FBFS)

File-based Feeding Strategy (FBFS)

The file-based feeding strategy is similar to the previously described feeding strategy using buffers. Instead of keeping the chunks in memory, a chunk-file per node is created on disk. As soon as a file reaches the size limit, it is transmitted to the receiving node. As Figure 5.8 shows, the process is similar to the BBFS. In comparison to the WBFS which transmitted the whole database at once, it allows for better pipelining as the data is now split up into several files which are transmitted consecutively.

5.4.3 Optimizing Imports at Receiving Nodes

Database vendors and implementations have been very active in order to improve the performance of their bulk loading tools as this is an important and critical step e. g., during the data staging into a data warehouse for business analytical processing. Therefore the performance of these tools is superior to a series of individual *insert*-statements or even several blocked *insert*-statements.[1]. Therefore, once data is transmitted to the receiving node, it is converted into a CSV file and the database load utility is used for import. So whenever multiple rows are inserted into a database, we use the load utility of the database. In general, of course, it is possible to adapt the import strategy to the amount of data which needs to be imported.

[1]A blocked SQL *insert*-statement combines several tuples in a single command, i. e., *insert into values (...), (...)*.

Parameter	Value(s)	Description
f	TBFS, WBFS, BBFS, FBFS	Feeding strategy
P	P_{obs}	Data set
p	262 144	Size of the partitioning scheme
n	16, 32	Network size
u	initial load, replication	Feeding use case
c	512 KB, 4 MB, 32 MB	Chunk size

Table 5.1: Parameters for the evaluation of feeding strategies

5.5 Feeding Throughput Evaluation

In the following, we present the evaluation results for the different push-based feeding strategies introduced in this chapter.[1] Table 5.1 summarizes the evaluation parameters. As use cases for our evaluation, we selected the initial load scenario and the scenario for replicating all data. Based on the observational data set P_{obs}, we created a partitioning scheme with 262 144 partitions. We used two network setups: The first used 16 nodes from our lab and the second setup consisted of 32 nodes from the AstroGrid-D and the PlanetLab test bed. For the chunk-based techniques, we used chunk sizes of 512 KB, 4 MB, and 32 MB in order to evaluate the importance of the chunk size for the feeding process.

5.5.1 Initial Load Evaluation

In the following, we present runtime results for initially loading data into our community-driven data grid.

Running Time for the “Wolf”-based Feeding Strategy

For the “wolf”-based strategy, we ran the export of the three observational catalogs in parallel. Hence, the biggest archive determined the complete running time for the initial step. After creating the CSV files, we combined these files into a database for each node. This complete database was finally copied to the participating nodes either with *scp* or *GridFTP*. The extraction of the CSV files lasted 1 hour for the SDSS archive and 30 minutes for the ROSAT and TWOMASS archives, each. As we extracted the data in parallel, this step lasted 1 hour and the data was extracted with roughly 23 000 tuples per second. Copying the data to the nodes, lasted up to 4 hours (depending on the node bandwidth). Finally, the creation of the node databases lasted about 3 hours, depending on the total number of nodes. Therefore, the whole process required about 8 hours in total.

Running Time for the Tuple-based Feeding Strategy

For the tuple-based implementation, we set the frequency to 1 000 tuples per second. This configuration was necessary in order to keep the underlying FreePastry implementation from

[1]The TBFS, BBFS, FBFS, and the pull-based strategy have been integrated into the HiSbase prototype. Only the WBFS is realized as an external program because we mostly use it to prepare databases offline, e. g., for the PlanetLab nodes.

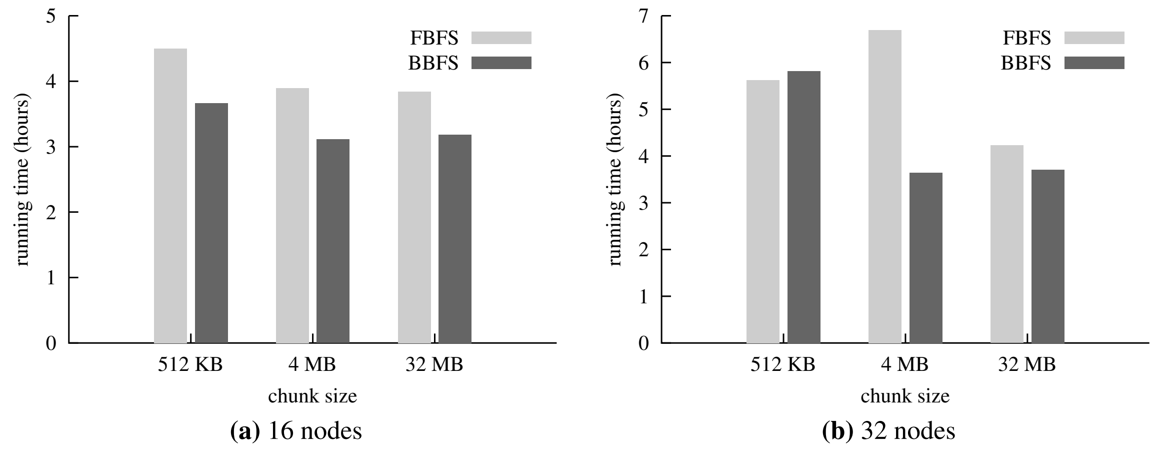


Figure 5.9: Results for the initial load scenario

dropping messages. Based on the fixed frequency we can compute the actual duration. The duration for our initial load scenario was about 23 hours. Crainiceanu et al. (2007), for example, used an insertion rate of 4 items per second during their experiments. However, feeding our SDSS sample (80 million objects) at that speed would last 33 weeks.

Running Time for the Chunk-based Feeding Strategies

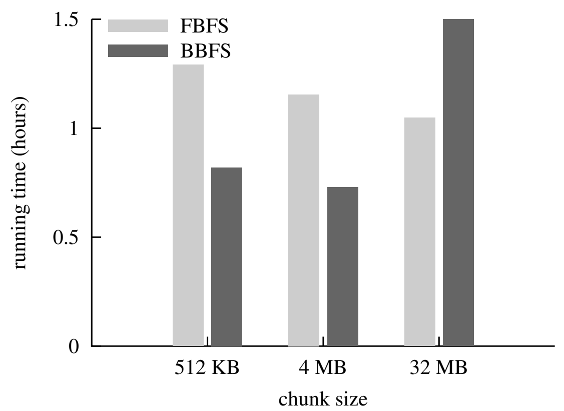
Figure 5.9(a) shows the results for the buffer-based (BBFS) and file-based (FBFS) strategies on our 16 nodes setup with different chunk sizes. Both strategies are faster than the WBFS as now all tasks (extraction, distribution, and data import) are performed in parallel. In addition, initial feeding is accelerated by using chunk sizes in the megabytes compared to the 512 KB chunk. However, the difference between 4 MB and 32 MB was not significant. The BBFS creates buffers in memory and submits them as serialized messages to the receiving nodes. Creating the CSV files for the database import is executed on the receiver side. Therefore the feeders can generate the next chunk faster and BBFS outperforms FBFS regardless of chunk size by approximately one hour. The results for the 32 nodes are similar though some measurements (especially those for the FBFS with 4 MB chunks) were distorted due to high loads on PlanetLab nodes which run several experiments in parallel.

5.5.2 Replication Evaluation

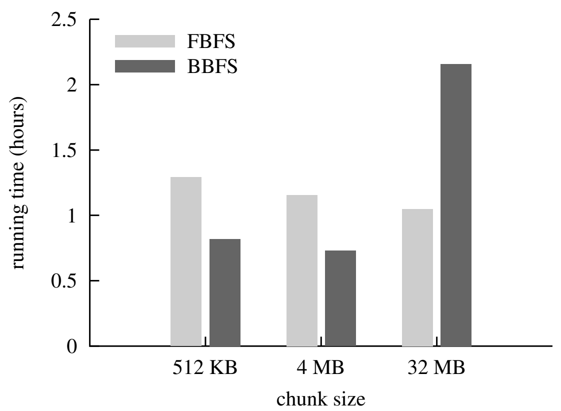
For the evaluation of the replication scenario, we used the databases created during the previous experiment as data sources. In order to ensure that all nodes redistribute their data, we assigned nodes with new key space identifiers.

Running Time for the Tuple-based Feeding Strategy

For the TBFS, we estimated the runtime using the largest databases on individual nodes. Of course, the exact size varies depending on the setup. Using the 1 000 tuples per second from the initial load scenario, our network with 16 (32) nodes was replicated in 6 hours and the 32-node network in 3.5 hours.



(a) 16 nodes



(b) 32 nodes

Figure 5.10: Results for the replication scenario

Running Time for the Chunk-based Feeding Strategies

From Figure 5.10(a) we see, both strategies greatly reduce the overall running time to 1 hour due to the increased parallelism. Again, the buffer-based strategy is slightly better, however both strategies are now more comparable. The results for 32 nodes, shown in Figure 5.10(b), are in accordance with those of our local setup. The result for 32 MB chunks of the BBFS must again be attributed to the high load on the PlanetLab resources.

5.5.3 Discussion

To summarize, the buffer-based strategy outperforms the file-based strategy in our measured scenarios as it introduces further parallelism by migrating tasks to receiving nodes. This is especially beneficial for settings where only a small number of sources feed their data into the system. Based on the current results, we suggest to use a chunk size of 4 MB. The differences between the measurements with 4 MB and 32 MB, respectively, were not significant and feeders complete 4 MB chunks faster. We plan further experiments with chunk sizes in the gigabytes and currently redo the experiments with 32 nodes in a setup with dedicated resources.

5.6 Related Work

P-Ring (Crainiceanu et al., 2007) and online load balancing (Ganesan et al., 2004a) are examples for extremely scalable P2P based infrastructures which achieve good load balancing for dynamic data distributions. Both show that two operations—either splitting existing data partitions or moving them—are required in order to achieve provable load balancing guarantees for data load balancing. However, the fact that they perform tuple-based insertion introduces an overhead which is too high for data-intensive e-science workloads.

Silberstein et al. (2008) propose a bulk insertion technique for range partitioned data storage used in hosted data services, such as PNuts (Cooper et al., 2008). Similar to our work, they propose to use a initial phase in order to achieve a good data load balancing among the storage nodes to spread the insert load across multiple machines. During their planning phase, data samples approximate the actual data distribution and a mapping from partitions to nodes is computed. The authors show that it is NP-hard to find a perfect mapping (by defining the vector packing problem, a variant of the bin packing problem) and propose a technique that offers provable, approximative guarantees by repartitioning and moving partitions before data is actually inserted. Similar to the techniques proposed for P-Ring and for online load balancing, PNuts repartitions and moves data. An interesting concept are so-called staging servers, where new data is randomly partitioned to in order to sample data sets and increase the throughput during the insertion by parallelism. While HiSbase enables different applications to access the same data and thus enable cross-application data sharing, PNuts keeps different applications separated, e. g., offering users flexible service level agreements. As a final difference, PNuts maintains the complete data set at each cluster that participates within the distributed storage whereas our work partitions the data across institutional boundaries.

BitTorrent (Cohen, 2003) is probably one of the most prominent P2P infrastructures. Its design aims at improving the overall systems throughput by using not only the download capacity but also the upload bandwidth by participating nodes. In contrast to HiSbase, however, BitTorrent clients, which cooperate in a download session, download the complete content. In HiSbase, we partition the content across multiple nodes.

Related work in high-performance computing has also investigated how to improve the data distribution throughput. Proposals range from using TCP in combination with deadlock-free routing in order to increase the overall throughput (Hironaka et al., 2009) to shaping the data transfer rates by increasing the TCP buffer sizes and using several parallel streams (Yildirim et al., 2008).

Rehn et al. (2006) report on their experience for providing a data distribution service PhEDEx to one of the LHC experiments and what challenges still need to be addressed in order to make data transfer architectures robust and scalable. For full integration into Grid middleware, the method of choice for data transfers is GridFTP (Allock et al., 2005). It offers certificate-based authorization and authentication as well as striping data transfers across multiple channels. Gu et al. (2006) describe their distributed storage system Sector using a UDP-based protocol for large message transfers.

Query processing on radio frequency identification (RFID) data, such as tracking the paths of different items along a production line also benefit from bulk data transfers. Krompaß et al. (2007) have identified considerable benefits of bulk data staging compared to tuple-wise staging when transferring RFID reader events from on-line caches to a data warehouse.

5.7 Summary and Future Work

In this chapter, we presented several push-based data dissemination techniques that use data chunks in order to accelerate data feeding compared to tuple-based or pull-based data distribution. Interesting topics for future work includes further increasing the parallelism of the initial feeding phase, e. g., using several staging servers as in PNuts. Improving the handling of border data when multiple feeders send the same catalog is another optimization which could yield interesting insights and benefits. Feeders, for example, could distinguish between original and border data. Once all sources have notified a receiving node that they have completed their feeding, these nodes only need to eliminate duplicates from the border data.

Efficient data dissemination is important for several scenarios within community-driven data grids, e. g., initially distributing data sets. Once all data sets are distributed, we can perform query processing and evaluate the throughput of our infrastructure, the topics of the following chapter.

CHAPTER 6

Running Community-Driven Data Grids

Within this chapter, we give more details of the query processing techniques of community-driven data grids. We present detailed query processing algorithms and discuss various query coordination techniques applicable for e-science communities (Section 6.1). In Section 6.2, we present the settings for throughput evaluations of our prototype and demonstrate the increased query throughput compared to centralized data management.

6.1 Query Processing

Due to the distributed nature of e-science collaborations, query processing will require a node which coordinates the common effort of data nodes that manage data which is relevant for a specific query. We denote this node as *coordinator* and it will collect intermediate results in order to perform the post-processing, e. g., duplicate elimination or further filtering. Depending on which node coordinates the query processing, some things might change: the amount of data which is needed to be transmitted, the distribution of the coordination load, the response time for the query processing. We will discuss in detail *what design choices are relevant for a good strategy to coordinate the distributed query processing in community-driven data grids.*

We describe the various design options for selecting a node as coordinator during query processing within community-driven data grids. First, we describe the data access patterns and algorithms of our query processing in Section 6.1.1. We then define five coordination strategies (Section 6.1.2) that either choose the node that submits the query or a node which contains relevant data. In our analysis (Section 6.1.3), we evaluate how many messages are required by each strategy, how much traffic is generated, and how the strategies level the coordination load among the nodes involved in the query processing based on results from simulation runs. Our results show that coordination strategies which use the relevant data nodes as coordinators achieve a good trade-off between fair load-balancing and reducing data-shipping during the query coordination independent of skew within the workload submission. Besides that, also the number of messages is significantly lower for such coordination strategies. Only for workloads with low coordination load, selecting the submitting node as coordinator is a feasible choice.

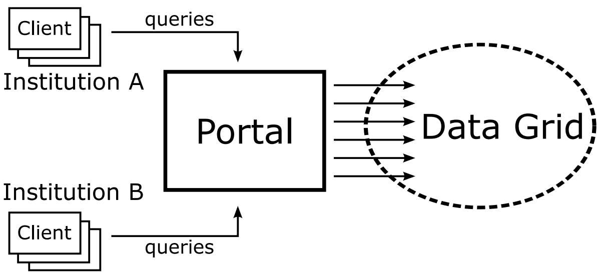


Figure 6.1: Portal-based query submission

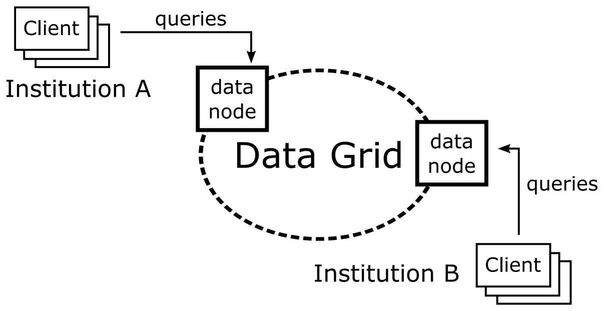


Figure 6.2: Institution-based query submission

Algorithm 6.1: submitQuery

```
Input: Histogram $h$, query $q$, submit node $s$
Output: query processing is initiated
  Multi-dim. query area $a \leftarrow extractArea(q)$
  Set $R$ of relevant region ids $\leftarrow lookupArea(h,a)$
  Coordinator id $r_c \leftarrow coordStrategy.choose(R \cup \{s\})$
  if $r_c \neq s.id$ then
    send $PreparedQueryMsg(R,q,s)$ to $r_c$
  else
    $coordinateQuery(s,R,q,s)$
  end if
```

6.1.1 Data Access Patterns

Two access patterns are increasingly popular within communities for scientific data sets: *portal-based* and *institution-based* data access. Portal-based interfaces (Figure 6.1) are getting increasingly popular, because browsers are a well-known “tool” on every researcher’s workbench. Furthermore, the usability of such interfaces has been further increased using technologies such as AJAX which offers immediate feedback to the users and the barrier for non grid-savvy users is greatly reduced. Behind the scenes, the portal forwards the queries issued by the clients to the data grid. As a central component, the portal might be a potential bottleneck, especially if a service becomes increasingly popular. Therefore communities often employ institution-based data access, as shown in Figure 6.2. Provided that all data grid nodes offer query submission capabilities, institutional clients can directly connect to the data node local to the institution. This data node then initiates the query processing within the data grid. From a community’s perspective, query coordination strategies (as defined in Section 6.1.2) that perform well regardless of the submission pattern are clearly preferable. The community then can deploy (or even change) the data access interfaces most appropriate for their users.

Histogram-based Query Processing

For the following discussion of the query processing algorithms, we assume that all queries are region-based. Whenever a HiSbase node s receives a region-based query, it proceeds according to Algorithm 6.1. The query predicate defines a multi-dimensional query area A, and node s determines the set R of relevant data region identifiers with help of the histogram. The query coordination selection technique selects one node as *coordinator*. We consider as candidate

Algorithm 6.2: coordinateQuery (processPreparedQueryMsg)

```
Input: Histogram $h$, query $q$, coordinator node $c$, submit node $s$, set $R$ of relevant regions,
    map $P$ of pending results $[q.id \rightarrow$ [size: $e$, submit: $s.id$, results: $T_q$]]
Output: query is coordinated
    Set $L$ of local regions $\leftarrow \{r \in R | r$ covered by $c\}$
    $P.put(\ q.id,\ [|R|,\ s,\ \{\}]\ )$
    for all $r \in R \setminus L$ do
        send $PartialQueryMsg(q,c,s)$ to $r$
    end for
    Result set $T \leftarrow processLocally(q)$
    $processPartialAnswerMsg(T,|L|,q.id,s)$
```

Algorithm 6.3: processPartialQueryMsg

```
Input: receiving node $n$, query $q$, coordinator node $c$, set $R$ of relevant regions
Output: result of local database is returned to coordinator
    Set $L$ of local regions $\leftarrow \{r \in R | r$ covered by $n\}$
    Result set $T \leftarrow processLocally(q)$
    send $PartialAnswerMsg(T,|L|,q.id,s)$ to $c.id$
```

nodes either the submitting node s or any node which covers relevant data, i. e., a node covering regions whose identifiers are contained in R. The choice depends on the selection strategy as discussed later on. Either the submit node itself is coordinator and continues coordinating the query otherwise a *PreparedQueryMsg* is routed to the coordinator peer. The receiver of the *PreparedQueryMsg* also performs the coordinateQuery routine.

According to Algorithm 6.2, the coordinator collects intermediate results (if multiple nodes are responsible for the relevant data) and performs post-processing of the intermediate results. The query processing algorithm is designed such that the query is sent to each relevant region identifier. Thus, each peer keeps track of the queries it received in order to avoid multiple executions of the same query. This is achieved by storing a hash for each query that currently runs on a peer. The hash is based on the query, the submitting node, and the timestamp of the initial submission in order to avoid collisions between queries from different hosts. Furthermore the coordinator maintains a map P of query identifiers $q.id$ to a data structure which contains the number of expected replies e, the id of the submit node $s.id$, and a set T_q of all result sets already received. The query is only forwarded to those region identifiers that are not covered locally. Once the node has contacted all regions that are not covered locally, the node submits the query to its local database and processes its partial result.

All other data nodes proceed according to Algorithm 6.3 and return the local result set to the coordinator. They compute the number of locally covered regions as *weight* $|L|$ of the result set, which is used by the coordinator later on. The result set is the intermediate result from the local database. Finally, the coordinator is informed to update its query statistics.

Every time, a coordinator receives a *PartialAnswerMsg*, it updates the local mapping P according to Algorithm 6.4. It adds the new result set to the set of existing results T_q, and computes the number of remaining regions based on the weight of the answers received so far. Once, all partial answers are received, the coordinator performs the post-processing and sends the assembled result set T_{full} to the submit node, which can forward T_{full} to the client.

Algorithm 6.4: processPartialAnswerMsg

```
**Input:** receiving node $n$, result set $T$ of the partial answer, weight $w$ of the result set, query
  identifier $q.id$, map $P$ of pending results $[q.id \rightarrow$ [size: $e$, submit: $s.id$, results: $T_q$]]
**Output:** result $T$ is added to the appropriate query and if all results have been received the
  final result set $T_{full}$ is computed and returned to the submit node
  Map entry $m \leftarrow P.get(q.id)$
  $m.size \leftarrow m.size - w$
  $m.results.add(T)$
  **if** $m.size = 0$ **then**
    Result set $T_{full} \leftarrow performPostProcessing(m.results)$
    send $FullAnswerMsg(T_{full}, qid)$ to $m.submit$
    P.remove(qid)
  **end if**
```

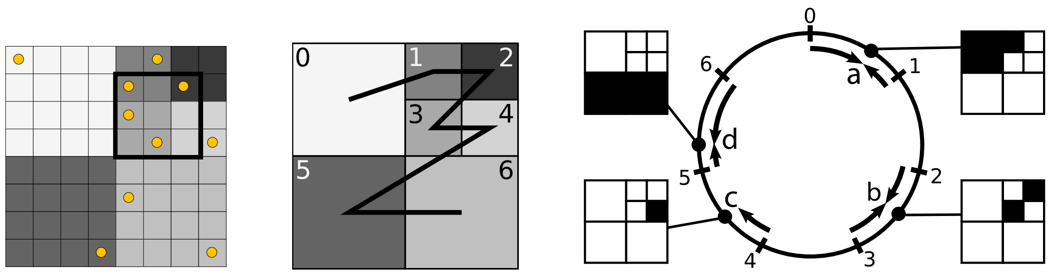


Figure 6.3: Example to illustrate query processing: (a) sample data set with exemplary query, (b) linearization of the histogram, and (c) the mapping of the regions to nodes

Example Query

For the following discussion of query processing, we use Figure 6.3 as a running example. The histogram regions (Figure 6.3(a)) are mapped to the identifier space preserving the order of the space filling curve (Figure 6.3(b)). Figure 6.3(c) shows a possible mapping of the seven partitions (0–6) to the identifier space of a HiSbase network with four nodes (*a*–*d*).

Let us assume that the query from Figure 6.3(a)—shown as the thick-lined rectangle—is submitted at node *d*. Node *d* analyses the received query and determines that regions 1, 2, 3, and 4 potentially contain relevant data. In the next section, we discuss now several design choices, whether node *d* itself or any of the nodes being responsible for relevant data (nodes *a*, *b*, and *c*) should coordinate the collaborative query processing.

6.1.2 Query Coordination Strategies

We now provide more details on the query processing within HiSbase. We especially discuss the various strategies to select a query coordinator that are provided by our framework.

In order to emphasize the various design options, we show the communication patterns for the various strategies. The used symbols are summarized in Figure 6.4. The node which performs the query coordination, i. e., either the submit node P_S or a node with relevant data D_i, is highlighted. We furthermore differentiate between messages that are routed via the underlying key-based routing mechanism (dashed arrows) or directly between nodes (continuous arrows).

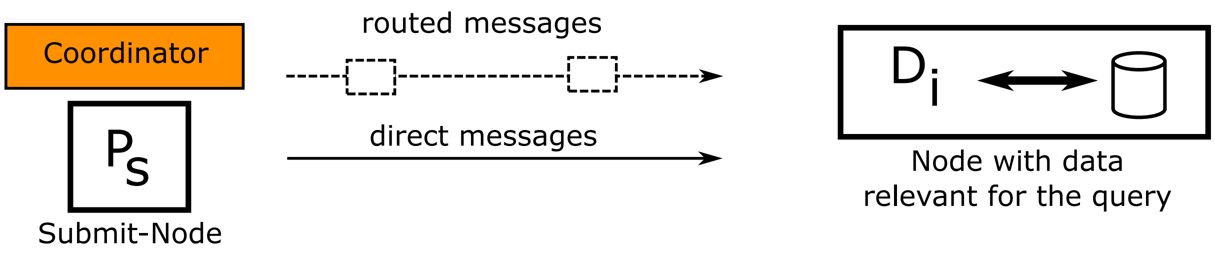


Figure 6.4: Key for the Figures 6.5 and 6.6

In large networks, routed messages may need several hops (indicated by the boxes on the arrow) as not all nodes store links to the other existing nodes. HiSbase only routes query messages via the overlay network, as such messages only contain a small payload. The payload of query messages consists of the SQL query itself and optionally additional information such as the relevant data regions. The results for those queries can be fairly large and therefore HiSbase always sends answer messages directly to the recipient. The recipients of answer messages can either be the coordinator (for building intermediate results) or the submitting node in order to return the result to the client. Of course, when the submitting node covers all the relevant data for a query, no additional communication on the HiSbase network is necessary. We now discuss five different strategies for choosing the coordinator.

SelfStrategy (SS)

The *SelfStrategy* chooses the submitting node as the coordinator, regardless whether it covers relevant regions or not. Figure 6.5 shows the message exchange if the SelfStrategy is used. We distinguish the two cases where either only one region—and therefore only one node (D_1)—contains relevant data (upper part) or the submitting node P_S needs to contact several nodes with relevant data. We start with the simple case first, which is that the query intersects only with one region that is covered by node D_1.

The submitting node P_S receives a *FullQueryMsg*. The node extracts the query area from the query predicate and determines the identifier of the relevant region from the histogram. As P_S is the coordinator, it sends a *PartialQueryMsg* to that region identifier and the message gets finally delivered to the data node D_1 which is responsible for the data region. D_1 submits the query to its local database and sends its *PartialAnswerMsg* back to the coordinator P_S. As D_1 was the only node which contained relevant data, the coordinator performs the required post-processing of the partial answer and sends the result back to the client using a *FullAnswerMsg*.

When multiple regions contain data relevant for the query, the coordinator routes a *Partial-QueryMsg* to all these regions. As in the simple scenario, each node submits the query to its database and returns the result to the coordinator. The coordinator then combines the intermediate results, performs any post-processing, and finally returns the result back to the client.

When the SelfStrategy is used for our running example from Section 6.1.1, node d performs the role of the coordinator. It routes *PartialQueryMsg*s to the regions 1, 2, 3, and 4 in parallel. Via the underlying key-based routing mechanism nodes a (for region 1), b (regions 2 and 3), and c (region 4) receive the messages and submit the query to their local databases. Finally, they send their answers directly back to node d. Please note that node b only sends one answer back to d, although it received two partial queries. Instead, node b informs node d that its result has a weight of two as it is based on two data regions.

The SelfStrategy achieves a good balancing of the query coordination load across multiple

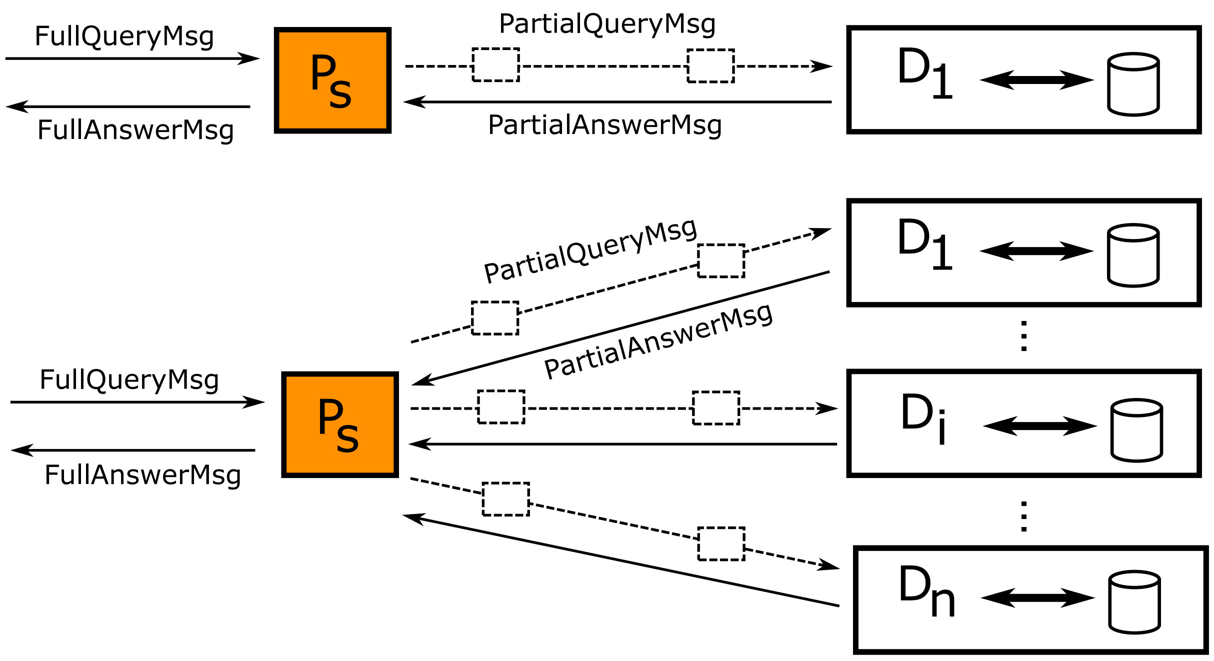


Figure 6.5: Message exchange for coordination strategies where the submitting peer (P_S) is the coordinator

nodes as each node is responsible for its own queries, when used with institution-based data access. Moreover, it distributes the coordination load in a fair manner, i. e., the more queries a node issues into the network the more it needs to coordinate. With portal-based data access, the portal node needs to do all the query coordination. The routing of all query messages is done via the underlying key-based routing mechanism. In a network with n nodes, it therefore potentially involves $O(\log n)$ messages in order to route messages to a node even though the region-based queries have a high locality. With increasing network size, this issue becomes a serious drawback with regard to the number of messages. Moreover, for queries covering multiple regions, all intermediate results need to be shipped to the submit node.

Motivated by these observations, all the strategies described in the following have in common that they select a node maintaining relevant data as coordinator. We therefore call them *region-based strategies* in the following. We do not consider selecting a random node from the whole network. The result of such a strategy would be counterproductive, as now *all* relevant data need to be transmitted to the coordinator and then to the submission node.

FirstRegionStrategy (FRS)

The *FirstRegionStrategy* always picks the node responsible for the first data region as coordinator. This strategy is representative for strategies that choose a fixed (e. g., the first, the last) region for coordinating the query processing in a simple way.

Figure 6.6 shows the according communication patterns. Like with the SelfStrategy, we first discuss the case, when the query intersects only one data region and thus only one node contains the relevant data. The submitting node P_S looks up the relevant data region in its histogram. Instead of coordinating the query processing itself, it now routes a *PreparedQueryMsg* towards node D_1, which covers the relevant data region. The node performs the local database lookup, performs the post-processing, and sends the result as a *FullAnswerMsg* directly to P_S, which simply forwards the message to the issuing client.

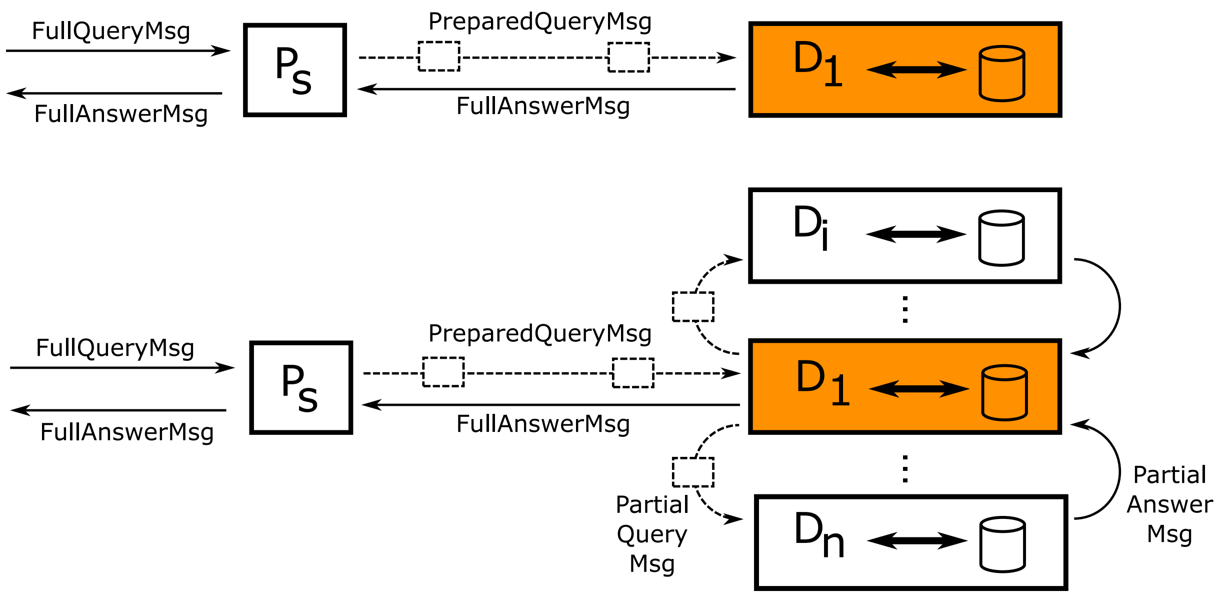


Figure 6.6: Message exchange for coordination strategies where a region with relevant data (D_1) is coordinator

If the query area intersects multiple relevant regions, the coordinator submits additional *PartialQueryMsgs* to the other data regions and receives the *PartialAnswerMsgs* from the according nodes, performs the post-processing, and collects them into a *FullAnswerMsg*.

For our running example query, node *d* would send the submitted query together with its relevant regions to region identifier 1, as region 1 is the region with the lowest region identifier containing relevant data. Therefore, node *a* becomes the coordinator and informs nodes *b* and *c* about the query. Having received the intermediate results from its local database and from both other nodes, node *a* delivers the final result to node *d*.

As noted in the previous section on the SelfStrategy, collaborating nodes are neighbors on the identifier space with high probability. This is due to the use of a space filling curve for mapping the data regions on the identifier space and because the high degree of spatial locality of the queries. Thus, the coordinator knows the direct link to the other data regions and can route the messages directly. As our evaluation in Section 6.1.3 shows, this can significantly reduce the number of messages during query processing. Moreover, this strategy benefits from the coordinator covering some relevant data which needs not to be transmitted for post-processing. This benefit is common to all region-based strategies.

SelfOrFirstRegionStrategy (SOFRS)

The *SelfOrFirstRegionStrategy* is a combination of the both previously discussed strategies. Whenever the submitting node covers data relevant for the query itself, it performs the query coordination. Otherwise, the node covering the first region is selected as coordinator.

When issuing our example query, e. g., to node *c* (instead of node *d*) using SelfOrFirstRegionStrategy, node *c* would coordinate the query itself.

The incentive for using the SelfOrFirstRegionStrategy is that if data is already at the submitting node it would be unreasonable to send that data across the network and returning it after post-processing. However, the number of regions that an individual node covers decreases with growing network size when keeping the partitioning scheme fixed. Thus, it becomes less likely that a submitting node covers any relevant regions itself.

CenterOfGravityStrategy (COGS)

While looking for similar ways to minimize the amount of data shipping like the SelfOrFirstRegionStrategy does, the *center of gravity*, the average location of the weight of an object, seems to be the "perfect spot" for minimizing the data transfer. Therefore, we define the *CenterOfGravityStrategy* which selects that node as coordinator that covers the partition which contains the center of gravity for an query area. As regular-shaped query areas (such as circles or rectangles) are very popular for specifying query areas in many scientific domains (e. g., the cone-searches in astrophysics) this approach is viable. Assuming, the relevance of data within intersected partitions is proportional to the intersection area between partition and query, the regions in the center of the query area cover more relevant data. Regions at the border of the query area are not completely covered and not that relevant under this assumption.

For our example query, the center of gravity lies within region 3, thus node b is selected as coordinator. Once node b has received the intermediate results of peer a and c, it creates the final result and sends it to peer d. In this case, the coordinator even covers most of the relevant regions (region 2 and 3) which is also good for reducing the amount of data to be shipped.

Unfortunately, the computation of the center of gravity can become quite complex or the region containing the center of gravity might not even be within the query area (e. g., for an O-shaped query area).

RandomRegionStrategy (RRS)

The final strategy presented in this section is the *RandomRegionStrategy*. As its name suggests, this strategy randomly selects one of the identifiers of the relevant data regions to determine the coordinator. For the example query any of the nodes a, b, or c could be selected as coordinator by this strategy. Note, however, that the probability of node b becoming coordinator is twice the probability of node a and c, respectively. The selection process is based on the number of relevant regions and therefore nodes covering more regions are more likely to be coordinators.

The strategy reduces the amount of data shipping between collaborative nodes, it only retrieves the relevant data regions and it is applicable for query areas of any shape. By design, it achieves a good trade-off between balancing the coordination load multiple nodes (as each node having relevant data can be selected) and reducing the data shipping for query coordination (as it prefers nodes with many regions as coordinator).

6.1.3 Evaluation of Query Coordination Strategies

For our evaluation, we used HiSbase, our Java-based prototype for community-driven data grids. As histogram data structure we used a quadtree-based partitioning scheme with 262 144 partitions as defined in Chapter 3. The data set, P_{obs}, consists of 137 million objects from subsets of the ROSAT, SDSS, and TWOMASS archives as described in Chapter 1.

The network size was varied from 100, 300, 1 000, to 3 000 nodes in order to evaluate the scalability of the five strategies described in the previous section. In the experiments, we use two query sets. The first query set, Q_{obs}, comprises about 100 000 queries and was constructed from real queries issued on the SDSS web interface. The second workload, Q_{scaled}, was created by scaling all queries of Q_{obs} whose search radii were among the five most frequent search radii to cover an area of one square-degree. Thus a significant fraction of the workload Q_{scaled}

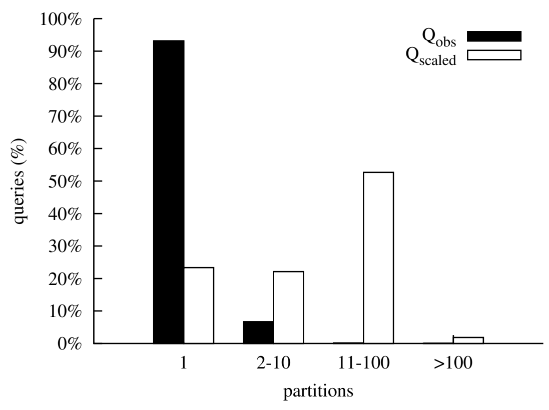


Figure 6.7: Percentage of queries that require data from different number of partitions

Parameter	Value(s)	Description
c	SS, FRS, SOFRS, COGS, RRS	Coordination strategy
n	100, 300, 1 000, 3 000	Network size
P	P_{obs}	Data set
Q	Q_{obs}, Q_{scaled}	Query workload
s	s_p, s_i	Submission characteristic

Table 6.1: General parameters for the evaluation of coordination strategies

will intersect multiple regions. Figure 6.7 shows that while more than 90 % of the workload Q_{obs} can be answered with data from a single data region (1.2 on average), the majority of the queries in Q_{scaled} require between 10 and 100 regions (average: 20 regions). Please note that the scaled workload has still a reasonable degree of locality, as 100 regions are 0.03 % percent of all histogram regions. As submission characteristics we used the both variants discussed in the beginning of Section 6.1.1, i. e., we submitted all queries from a single node (the portal-based characteristic s_p) or submitted the queries from a randomly selected node in the network (the institution-based characteristic s_i). Table 6.1 summarizes the general parameters of the evaluation setup.

The results in this section are obtained from running HiSbase instances on the discrete-event simulator provided by FreePastry. The simulator environment allows to fix the assignment of nodes to the identifier ring. Thus we were able to use the same mapping for all five strategies in order to directly see the difference due to the strategy and not due to a different region-to-node mapping. As a particular region assignment might give advantage to one of the strategies under evaluation, we created 10 different mappings in total. Measuring 5 strategies, 4 network sizes, 2 workloads, and 2 submission characteristics results in 80 distinct configurations leading to 800 individual measurements. These individual measurements were conducted on the Linux cluster of the Leibniz-Rechenzentrum (LRZ), using the Globus job submission of the D-Grid interfaces.

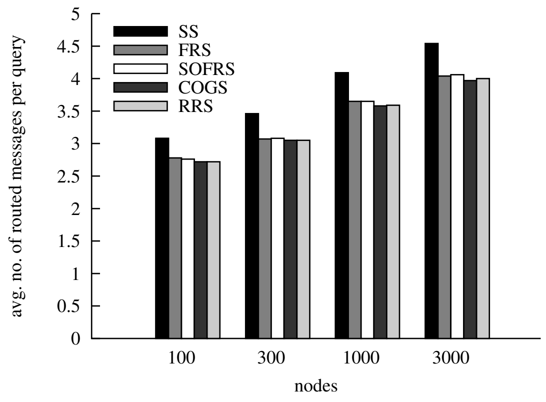


Figure 6.8: Average number of routed messages per strategy with workload Q_{obs} and portal-based query submission (s_p)

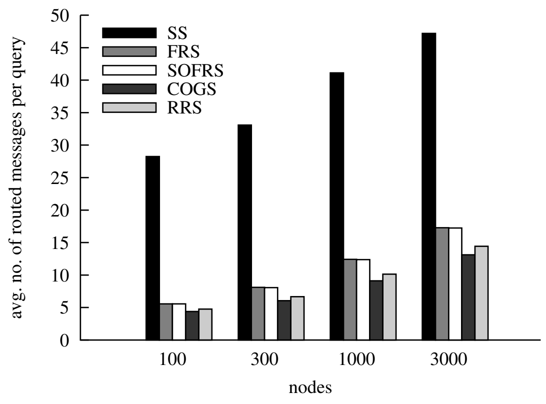


Figure 6.9: Average number of routed messages per strategy with Q_{scaled} and portal-based query submission (s_p)

Query-related Messages

The average number of messages per query is the first aspect we investigated for each query coordination strategy. We consider a strategy to be better, when it requires less messages during query processing.

Figure 6.8 shows the average number of query-related routed messages for Q_{obs} using a single node for submitting all queries (s_p). For any network size, all coordination strategies which select data regions as coordinators need almost the same amount of messages. This is clearly caused by more than 90% of the queries intersecting only with one data region. In this case, all region-based strategies are identical. The SelfStrategy requires additional 0.5 messages on average due to the remaining queries which intersect more than one region. The strategy routes the partial queries from the submission host to the data regions (see Figure 6.5) and therefore it is likely that *all* these messages require multiple hops to reach their destination. In small networks, there is a small probability that the submitting node P_S already contains all relevant data, i. e., P_S and D_1 are the same, and no messages are required. This explains the 2.7 messages on average for the region-based strategies for a network with 100 nodes.

For the scaled workload Q_{scaled}, the messaging overhead of the SelfStrategy becomes even more evident, as shown in Figure 6.9 for the portal-based approach. In Figure 6.10 we submitted the queries from random nodes. Interestingly, all five strategies use the same amount of

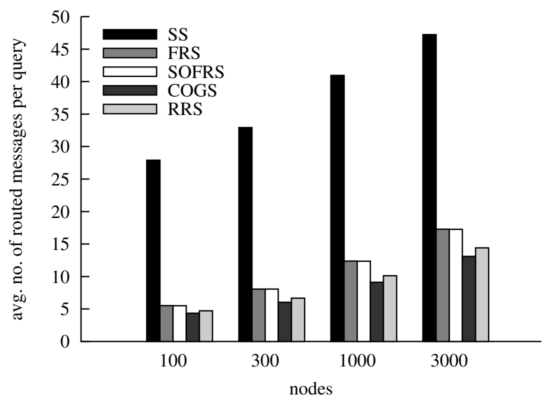
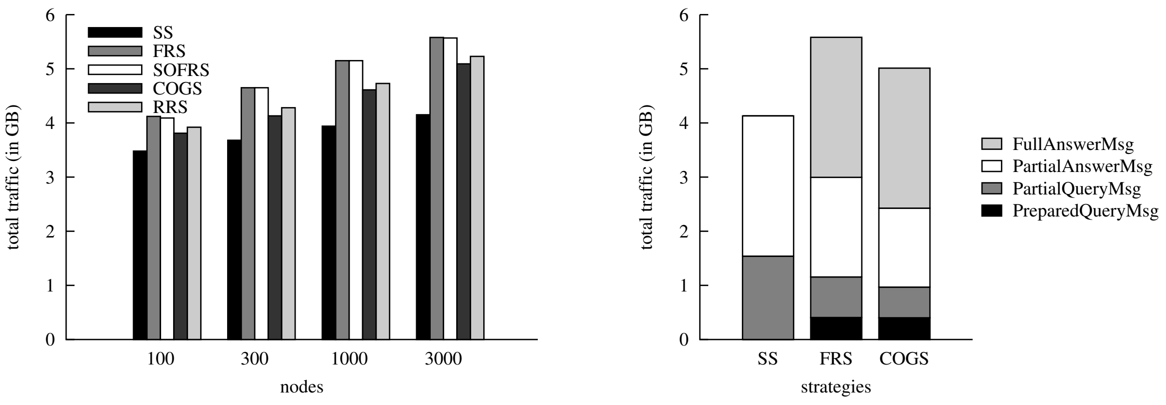


Figure 6.10: Average number of routed messages per strategy with Q_{scaled} and institution-based query submission (s_i)



(**a**) total network traffic (**b**) traffic details by message type

Figure 6.11: Network traffic statistics for workload Q_{obs} and institution-based query submission (s_i) about (a) the total traffic by strategy and (b) the traffic by message type for a run with 3 000 nodes

messages per query in both submission scenarios.[1] The statistics only show the perspective of the complete network and therefore existing message skew on the network connections is not visible. Furthermore, the results of Q_{scaled} reveal that the SelfOrFirstRegionStrategy predominantly uses its "FirstRegion"-part, due to the observation in Section 6.1.2, that it is unlikely for the submitting nodes to cover relevant data itself in large networks. Finally, both the CenterOfGravityStrategy and RandomRegionStrategy perform best for the scaled workload in terms of query related messages. Both strategies are able to reduce the number of required messages as anticipated.

Query-related Network Traffic

Besides the number of messages, we also compared the actual message traffic of the various strategies in our simulation environment. Figure 6.11(a) shows that SelfStrategy has the lowest overall traffic compared to all region-based strategies. This can be attributed to the fact that the coordinator does only append the intermediate results and does not perform further data reduc-

[1]This was also the case for Q_{obs} and s_i.

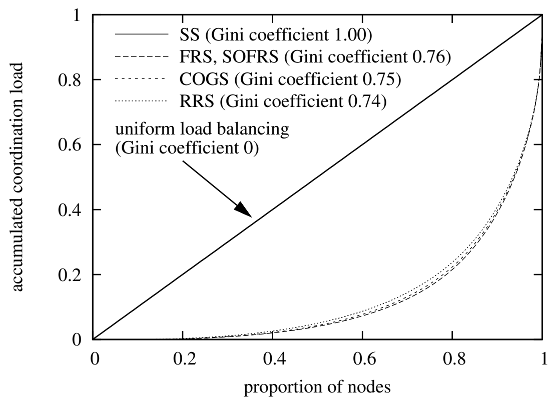


Figure 6.12: Lorenz curves of the coordination load distribution on 3 000 nodes and portal-based query submission (s_p)

tion. Therefore, for region-based strategies both *PartialAnswerMsg*s *and FullAnswerMsg*s are transmitted.

When splitting the traffic into the constituents by the different message types, as shown in Figure 6.11(b), we see the major trade-offs between using the SelfStrategy or region-based strategies such as FirstRegionStrategy or CenterOfGravityStrategy. Region-based strategies reduce the traffic during the query initialization phase, i. e., for both region-based strategies the combined traffic of *PreparedQueryMsg*s and *PartialQueryMsg*s is less than the traffic of *PartialQueryMsg*s sent by the SelfStrategy. By choosing the coordinator region wisely (CenterOfGravityStrategy), we can reduce data shipping to the coordinator (*PartialAnswerMsg*s). However, the overall traffic for region-based strategies increases when the *FullAnswerMsg* is built by appending the intermediate results. As a consequence, the traffic by *PartialAnswerMsg*s for the SelfStrategy is equal to the traffic by *FullAnswerMsg*s from region-based strategies.

Coordination Load-Balancing

The last aspect we analyzed in our simulations was the ability of each strategy to balance the coordination load evenly across multiple nodes. The more nodes are involved the more evenly the coordination load is distributed. We evaluate the uniformness of the load distribution using the Lorenz curve and the Gini coefficient (see Pitoura et al., 2006). In our case, the Lorenz curve accumulates the queries coordinated by all HiSbase nodes in ascending order, i. e., it starts with nodes which do not coordinate queries and finishes with the node which has the highest load. The Gini coefficient is defined as the area between the Lorenz curve for the distribution and the diagonal. The lower the Gini coefficient, i. e., the closer the graph to the diagonal, the more uniform the load is distributed among the nodes. The final accumulated coordination load is the complete size of the query workload.

In the setup with portal-based data access, the SelfStrategy does not perform any load-balancing as all queries are coordinated by the single submitting node. This results in a Gini coefficient of 1 and therefore the curve is hardly visible in Figure 6.12 as it coincides with the x-axis and the y-axis on the right-hand side. The coordinator selection strategies based on data regions have Gini coefficients of around 0.75, the RandomRegionStrategy and CenterOfGravityStrategy being slightly better than the other two strategies. When submitting queries from random nodes (Figure 6.13), the SelfStrategy almost evenly distributes the coordination load while the level of load-balancing of the other strategies remains the same.

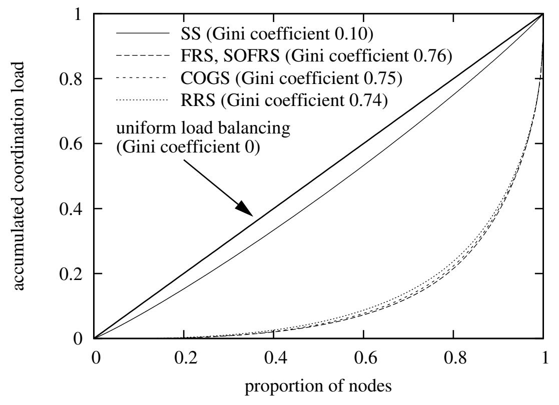


Figure 6.13: Lorenz curves of the coordination load distribution on 3 000 nodes and institution-based query submission (s_i)

For both workloads, only a fraction of the histogram partitions (e. g., only 25 % for the Q_{obs} workload) receives any query, i. e., both workloads exhibit query hot spots. Distributing the coordination load randomly on all network nodes is not desirable as it would ignore the data locality during the query processing. However, mitigating such query hot spots by replication using techniques which we will discuss in Chapter 8 opens the opportunity for more advanced coordination strategies.

Discussion

Based on the results obtained during the evaluation presented in this section, we gained several insights. All region-based coordination strategies reduced the number of messages considerably compared to the SelfStrategy and performed independently from the submission characteristics. Especially, the CenterOfGravityStrategy and the RandomRegionStrategy even further reduce the required messages by preferring those nodes as coordinator, which cover a significant part of the relevant data. Whereas the CenterOfGravityStrategy required less messages than the RandomRegionStrategy, the latter does not have the restrictions of the CenterOfGravityStrategy as discussed in Section 6.1.2 (computational complexity, shape of query area).

With regard to network traffic, the SelfStrategy is better suited for queries, when merging the intermediate results is the predominant part of the post-processing at the coordinator. However, when the data is further reduced by the coordinator (e. g., in queries with group-by and having-clauses), a region-based strategy is preferable, as only the relevant data is transmitted to the submit node.

6.1.4 Summary and Future Work

Region-based queries require the selection of a *coordinator* during the distributed query processing within community-driven data grids, i. e., a node which coordinates the collaboration of data nodes to extract the relevant data and which performs post-processing.

In this section, we have defined several strategies, that select the query coordinator from the node submitting the query or from the nodes covering relevant data. We evaluated the query coordination strategies on a data set from astrophysical observations with actual application workloads using two of the dominant query submission patterns within scientific communities like portals or institution-based data access. In our simulation studies we compared how many messages are sent and how much network traffic is generated by each strategy and how they balance the coordination load. The results show that strategies that aim at reducing the amount of data transmitted to the coordinator (CenterOfGravityStrategy and RandomRegionStrategy) offer the best trade-off between both criteria independently of the submission characteristic. If coordination load is low (the coordinator performs no complex filtering tasks) then the SelfStrategy is the strategy of choice.

In the results of the coordination load-balancing, the influence of query hot spots has been clearly visible. In order to mitigate query hot spots at runtime, one approach is to replicate the popular data regions to multiple nodes. To devise a coordination strategy that also considers replicated data regions during the selection process is an interesting issue for future research.

Furthermore, an interesting direction for further investigations is to analyze the individual queries in more detail and choose the coordination strategy based on structural properties and expected coordination workload. The results presented in this chapter provide a solid base for such developments.

6.2 Throughput Measurements

In the following, we evaluate the throughput performance of our system. We begin with an outline of the general definitions used during our throughput experiments. Initially, we compare a HiSbase instance with one node against a single database server and scale the instance to 16 nodes within our computing lab. In this setting, we achieve a super-linear throughput improvement. We then describe measurements conducted in the AstroGrid-D and PlanetLab test beds to discuss the impact of coordination strategies on the overall throughput. Based on the findings of this chapter, we identify the areas that offer further improvements for query throughput.

6.2.1 General Definitions

We measure throughput for varying *multi-programming levels (MPLs)*, i. e., a varying number of parallel queries in the system, to evaluate at what degree of parallelism a distributed architecture can outperform a centralized solution. Each run has k nodes, a batch containing l queries, and an MPL m. MPL=m denotes that *each* node keeps m parallel queries in the system. At the start of a run, each node immediately submits m queries. We measure the timestamp $s_{n,q}$ when node n has *submitted* its q-th query and the timestamp $r_{n,q}$ when it has *received* the corresponding results. After receiving an answer, nodes submit their next query in order to sustain their multi-programming level.

For measuring the throughput, we only consider queries processed in the time span when every node is guaranteed to work on MPL=m parallel queries, the *saturation phase I_{sat}*. The time interval between the point in time when the last peer has submitted its m-th query and the first peer has submitted its last query denotes I_{sat}, which is expressed formally as:

$$I_{sat} = \left[\max_{1 \leq n \leq k} (s_{n,m}), \min_{1 \leq n \leq k} (s_{n,l}) \right] \tag{6.1}$$

Let $\overline{I_{sat}}$ be the length of the saturation phase in seconds. The *throughput per second T* is based on the number of successfully processed queries during the saturation phase I_{sat}:

$$T = \frac{|\{(n,q) \mid r_{n,q} \in I_{sat}, 1 \leq n \leq k, 1 \leq q \leq l\}|}{\overline{I_{sat}}} \tag{6.2}$$

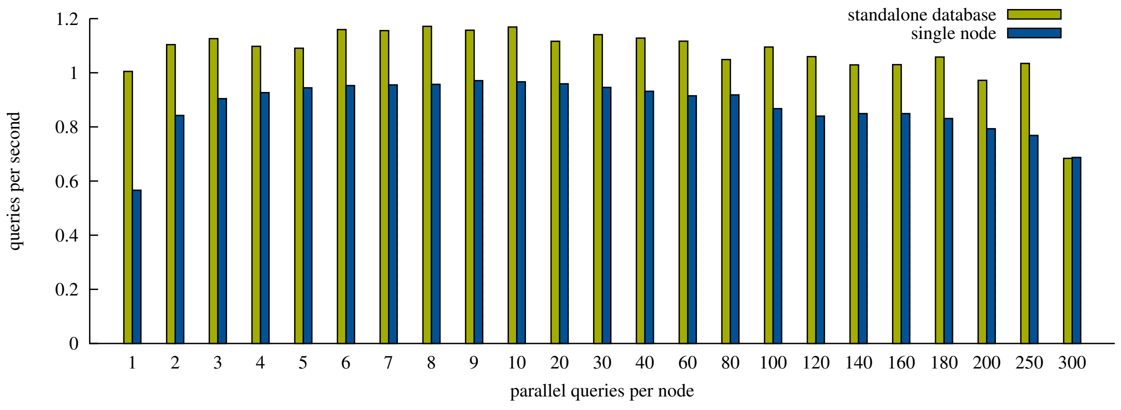


Figure 6.14: Query throughput results for the standalone database and single node configuration

We shortly illustrate the case of computing I_{sat} for one single HiSbase node n_1. Let MPL=10 and $l = 500$, then the node submits 10 queries to HiSbase in order to reach the desired degree of parallelism. In this scenario, I_{sat} starts at $s_{1,10}$. As soon as a query result is received, a new query is issued to HiSbase. Finally, the saturation phase I_{sat} ends when the node submits its last (500th) query at timestamp $s_{1,500}$. When multiple nodes participate in the HiSbase network, the last $s_{n,10}$ and the first $s_{n,500}$ timestamp determine the saturation phase of the complete network, as defined in Equation 6.1.

6.2.2 Evaluations in a Local Area Network

Throughput of a Single Node Instance We used a body of 730 *cross-match* queries for our evaluation. Cross-match queries determine whether data points from different sources are likely to stem from the same celestial object. The queries were created from 730 random sources of the SDSS catalog, using rectangular regions with an edge length of $0.05\,^\circ$. The size of the query rectangles is based on realistic values and each query covers approximately an area which is $\frac{2}{10^7}$ of the whole sky. Nodes submit these queries in random order. To this end, we present results for a quadtree-based histogram with 256 regions using the center splitting strategy.

The first experiment compares the query throughput of a standalone database with the query throughput of the same database used by a single HiSbase node to measure the overhead introduced by the HiSbase layer. The node is a Linux server with an Intel Xeon processor at 3.06 GHz, 2 GB RAM, and IBM DB2 V8.1. Queries to the standalone database are submitted via parallel JDBC connections. Figure 6.14 shows the throughput in queries per second of the standalone database and the single node HiSbase instance. The throughput increases for both single node setups through higher parallelism until their maximum throughput (sweet spot) is reached. The maximum throughput of both systems is roughly at 10 parallel queries: 1.17 queries per second at MPL=8 for the standalone database and 0.97 for the single node HiSbase instance at MPL=9. Although the standalone database performed better than the single HiSbase node in our evaluation, HiSbase introduces an acceptable overhead as in practice an instance with multiple (typically hundreds of) nodes is used.

Just to give an impression of current throughput figures, the traffic statistics of the SkyServer[1] archive show that between August 2008 and August 2009 about 13 000 queries per

[1]http://skyserver.sdss.org/log/en/traffic/

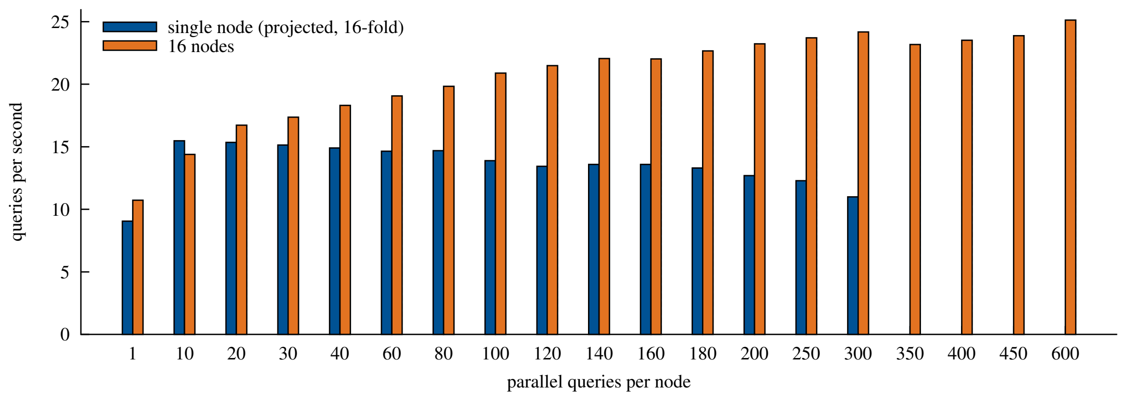


Figure 6.15: Throughput comparison of the multi-node instance with the projected values of the single-node configuration

month have been submitted to the SQL interface on average. This corresponds roughly to less than one query per second. However, there are already several occasions where the number of queries per second is significantly higher.

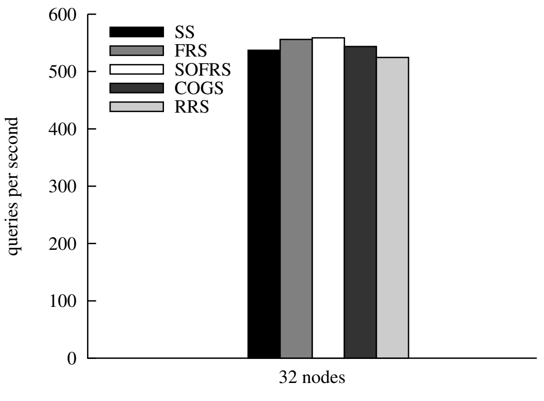
Throughput of a Multi-Node Instance We tested a multi-node instance in a local area network (LAN) which measures how HiSbase performs in a setting with low latency and high network bandwidth. The LAN configuration of HiSbase was set up on 16 consumer-class Windows PCs equipped with 1.6 GHz Processors, 512 MB RAM, and again with the IBM DB2 V8.1 database system. Figure 6.15 contrasts the projected throughput of the single node configuration described above (by multiplying the previous results with 16) and the 16-nodes instance. The 16 nodes achieve a stable super-linear throughput compared to the single peer from MPL=20 onwards. Less data on the individual node and especially a higher cache locality constitute this throughput improvement as nodes only process similar queries. We did not continue the measurements beyond an MPL=600, which corresponds to 9 600 parallel queries, as the expected numbers of parallel users are currently below this degree of parallelism.

6.2.3 Evaluations with AstroGrid-D and PlanetLab Instances

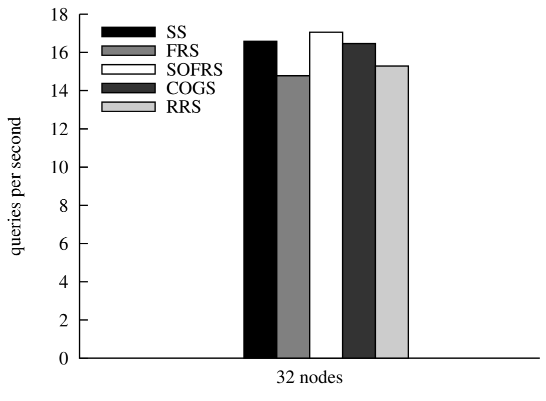
In order to verify the scalability of our HiSbase approach, we also conducted benchmarks on resources within AstroGrid-D and D-Grid as well as on the PlanetLab test bed, as PlanetLab is widely used for evaluating globally decentralized applications. In PlanetLab, applications run in so-called *slices* (virtual machines) and in parallel with several other installed applications. Within the AstroGrid-D test bed, the resources are more dedicated, reliable, and have high-bandwidth links. We successfully demonstrated HiSbase using up to 56 resources from our labs, the AstroGrid-D test bed, and on PlanetLab.

Impact of Coordination Strategies on Query Throughput The following throughput measurements were conducted on distributed HiSbase instances and complement the results for the coordination strategies SelfStrategy (SS), FirstRegionStrategy (FRS), SelfOrFirstRegionStrategy (SOFRS), CenterOfGravityStrategy (COGS), and RandomRegionStrategy (RRS) presented in Section 6.1.

We performed the experiments on a total of 32 nodes comprised by 11 resources from the AstroGrid-D test bed, 16 nodes from our computer lab, and additional 5 close-by resources



(a) Q_{obs}



(b) Q_{eval_scaled}

Figure 6.16: Throughput for 32 nodes with MPL=500 for (a) workload Q_{obs} and (b) workload Q_{eval_scaled}

from PlanetLab. After assigning each node with a random identifier, we distributed the data according to the same partitioning scheme with 262 144 partitions which was used during the simulation to evaluate the messaging overhead in Section 6.1.

As query workload we used a subset Q_{eval} of 22 000 queries from the observational workload Q_{obs}, which were among the top 20% fastest queries when run against a single DB2 instance containing all data. Those queries—with running times between 1 and 4 ms—would be penalized most severely from being submitted to a queuing system. Each node permuted the queries of Q_{eval} for its query batch. In order to measure the throughput for a scaled workload Q_{eval_scaled}, we applied the same scaling to the queries of Q_{eval} as in Section 6.1 to Q_{obs}. The queries in the scaled workload consider more data relevant for the cross-matching and therefore have a much longer running time. The throughput figures presented for this measurements are the averages built over three evaluation runs. Each node was equipped with an H2 database[1] for performing the local query processing. We decided to use an open source Java-based database for the evaluation, as it is easily distributed together with the application and does not require any administrative rights that are necessary to install most of the major database systems and it is free of licensing issues. We configured each HiSbase node to allow 10 parallel queries on its local database as suggested from the single instance comparisons above.

The throughput results for both workloads Q_{eval} (Figure 6.16(a)) and Q_{eval_scaled} (Figure 6.16(b)) show that for small network sizes all query coordination strategies are comparable. Performing experiments in a distributed environment is important to complement simulation results as they capture many side effects that can be left aside in simulations. However, all strategies showed a high variance in throughput across the different runs. To this end, we cannot determine a significant impact of the coordination strategy on the overall throughput for the evaluated HiSbase instance.

Impact of Increasing the Network Size During the evaluation in the AstroGrid-D test bed using the SDSS query log, the presence of query skew became evident. We illustrate this fact with Figure 6.17. We scaled the network size from 32 and 64 to 128 nodes and counted the top contributors, i. e., the nodes processing the most queries first, until we reached 90% of the

[1]http://h2database.com

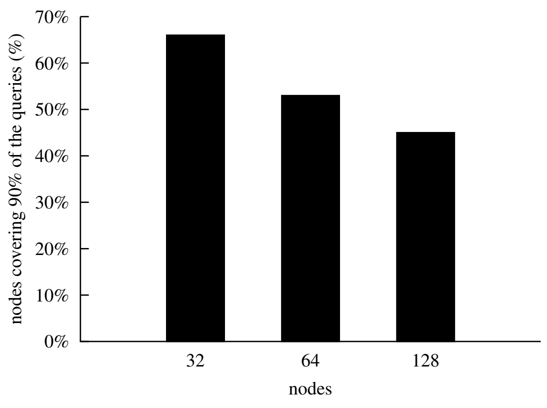


Figure 6.17: Fraction of nodes contributing 90% of the overall queries when intreasing the network size

queries. While 21 (66%) of the 32 nodes processed 90% of the queries, only 58 (45%) help on that query share in the 128-node network. The remaining 55% of the nodes were not used to their full capacity as they received only a small fraction of the queries.

6.2.4 Discussion

So far, community-driven data grids use several techniques in order to improve the query throughput within e-science collaborations: By partitioning the data space such that dense areas are partitioned more often, we can achieve data load balancing in community grids and reduce the amount of data for each individual node. By this parallelism, we can improve the query throughput. We can further preserve the query locality by using a space filling curve to map the partitions to the participating nodes and thus achieve a high caching effect as nodes receive only similar queries.

To further improve the throughput results from our experiments, we will now focus on query load balancing, as skewed query distributions resulted in a sub-optimal usage of available processing resources. Our techniques for increasing the workload-awareness during the training phase (partitioning creation) and at runtime are the subjects for the following two chapters.

CHAPTER 7

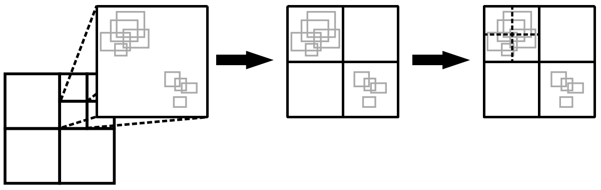
Workload-Aware Data Partitioning

In the preceding chapters, we have described the core building blocks of community-driven data grids and how the combination of these building blocks results in an increased throughput during query processing. To further improve the throughput gains by our data management infrastructure, we now discuss techniques to incorporate *workload-awareness* into community-driven data grids. In this chapter, we generalize the data-driven partitioning schemes of community-driven data grids to a cost-based partitioning in order to address two important challenges in scientific federations: data and query load balancing.

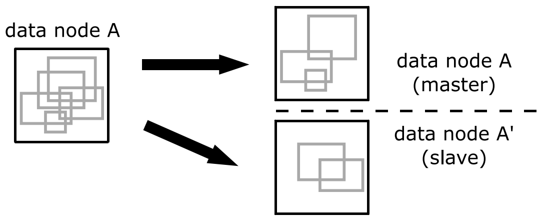
When some areas of the data space are very popular within the community, data nodes covering these areas, so-called *query hot spots*, tend to become the bottleneck during query processing. In general, query load balancing can be achieved via *splitting* (partitioning) and *replication* (Section 7.1).

For our cost model, we define several *weight functions* (Section 7.2). Besides data partitionings which consider only data load, we propose weight functions that allow advanced and *workload-aware* weighting schemes. One weight function, for example, combines the weight for points and queries to compute the *heat* of regions as the product of the associated data points and queries. Finally, we describe a weight function that decides by using the *extent of queries* whether replication is better than splitting a region.

In Section 7.3, we evaluate our approach using quadtree-based partitioning schemes as introduced in Chapter 3. We use a sample of one million queries from an SDSS query trace Q_{obs} on the skewed, observational data set P_{obs} and a synthetic workload on the uniform data sample P_{mil} from the Millennium simulation, as we have introduced in Section 1.2. We furthermore perform several throughput measurements on our local resources of the AstroGrid-D testbed as well as in a simulated network with the various partitioning schemes. The evaluation results assess the effectiveness and applicability of our load balancing techniques. We discuss related work in Section 7.4 and conclude this chapter in Section 7.5.



(a) Splitting (Partitioning)



(b) Replication

Figure 7.1: Balancing query load (gray query rectangles) via splitting and replication

7.1 Load Balancing Techniques

Query load balancing is a challenging task in distributed query processing. When dealing with popular ("hot") data, two strategies are generally applied in order to reduce the heat at the node predominantly responsible for the data:

Splitting (Partitioning) By further dividing the partition, parts of the query load can be moved to a different partition. If that partition is covered by another node, load is balanced between these nodes. If hot areas are distributed among multiple nodes, good load balancing is achieved.

Replication Sometimes migration is not possible (e. g., one single data object is "hot") or desirable (e. g., the query processing would result in more communication overhead). In such cases, load balancing can only be achieved by making multiple copies of the hot region at several locations. If all replicas participate equally during query processing, our design again achieves good load balancing.

Figure 7.1 shows the increased flexibility of load balancing techniques that apply both partitioning and replication. Initially, partitioning succeeds in dividing the two hot spot areas denoted by several gray query rectangles. The second partitioning step in Figure 7.1(a), however, would introduce additional communication overhead. In such situations, we can mitigate query hot spots better by replicating the original data area. Thus, multiple copies are available during query processing, as illustrated in Figure 7.1(b).

For deciding whether we gain more from replicating a region instead of splitting it, the following information is considered in our heuristics:

amount of data we still prefer to split those regions first that contain a considerable amount of data due to the importance of data load balancing,

number of queries regions with many queries should be handled before those regions whose workload is considerably low for query load balancing reasons,

extent of regions and queries in order to balance query load, we rather replicate regions whose workload predominantly consists of queries having a large *extent*, i. e., they cover a large area compared to the area of the region itself.

Figure 7.2 depicts the basic idea, why it is important to incorporate the relationship between the extents of regions and queries in a replication-aware weighting scheme. If all data and queries were uniformly distributed, a quadtree-based partitioning scheme with n regions has a maximum height of $\log_{2^d} n$ in the general case and $\log_4 n$ for our running example. If either

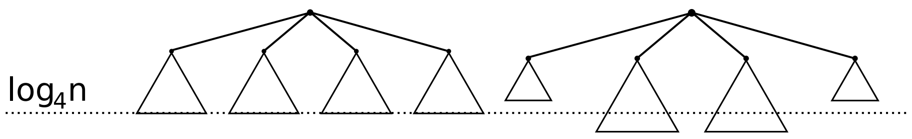


Figure 7.2: Impact of skew on the height of the leaves

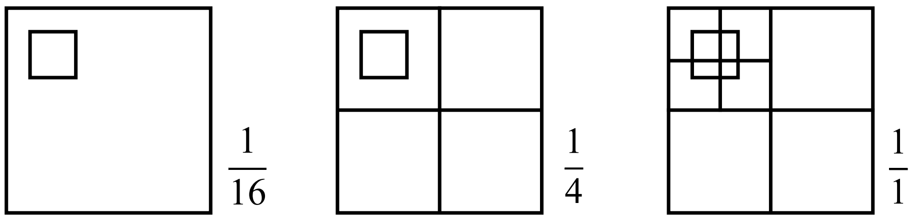


Figure 7.3: Impact of splitting a leaf on its ratio to a query area

data or queries are skewed, the regions with less "load" are leaves with height $h < \log_4 n$ (with a larger area) and those in a hot spot area have $h > log_4n$ (with a smaller area). A query area which covers about $\frac{1}{16}$ of a region, will cover $\frac{1}{4}$ on the next level and eventually will have the same size as a region after another split, as shown in Figure 7.3. Thus, the query is very likely to span multiple regions and to produce additional communication overhead.

In order to take the aspects just described into account, a weight function needs to adhere to the following heuristics. The weight function distinguishes regions with many data points from regions with only a few data points. Similarly, it regards workloads with many and few queries separately and moreover pays attention to whether queries have a small or large extent.

If a region contains *little data* and only a *few queries*, it should be neither split nor replicated as it does not contribute significantly to the overall load of the system.
In case of regions that contain *little data* which is interesting to *many queries*, we replicate those regions that have many big queries. These data partitions can be replicated at several nodes and then all replica are available for query processing. Partitions with many small queries are further split, as the resulting partitions possibly will fall into the category with few data points and few queries.
If a partition contains *many data points* but is only relevant to *few queries*, we prefer to split the partition. The performance of the small queries might increase as they run on smaller data sets which will even out the additional communication overhead for the big queries with regard to the overall performance.
The crucial class for the weighting scheme is the forth category of regions which contain *much data as well as many queries*. This category requires a good choice between splitting a region if small queries mainly constitute the workload and replicating the region if the ratio of big queries is higher.

Table 7.1 summarizes the options to either split or replicate regions based on their associated number of data points and queries.

After having described the intuition about our criteria for query load balancing, we now present our approach to create workload-aware data partitionings using weight functions.

Data points	Few queries		Many queries	
	Small	Big	Small	Big
Few	–	–	SPLIT	REPLICATE
Many	SPLIT	SPLIT	SPLIT	REPLICATE

Table 7.1: Categorization of regions for the replication-aware weight function

7.2 Region Weight Functions

In the following, we define a set of three weight functions for points, queries, and regions. This enables communities to create partitioning schemes for their data grids with a higher flexibility.

During the course of discussion, we will use several variables for points, queries, and regions which we will define in this paragraph. While creating our partitioning scheme, in the *training phase*, we use a representative *training data set P* and a *training workload Q*. The variable p denotes a data point from P and q is a query from Q. Note that scientific data sets often comprise many dimensions. For simplicity and ease of presentation, we only consider the projection to the most predominant attributes from the query patterns (such as the two celestial coordinates in our astrophysics example) during the training phase. When distributing the data partitions, the complete data is distributed.

The hyperrectangle A_q describes the boundaries of query q within the data space of the partitioning scheme. Likewise, we define the area A_r covered by the region r. These areas are important building blocks for our weight functions.

We consider data points as *relevant* for query q if they reside within A_q. We further denote the set of points relevant for query q as P_q and define Q_p as the queries for which p is relevant.

$$P_q = \{p \in P \mid p \in A_q\} \tag{7.1}$$
$$Q_p = \{q \in Q \mid p \in A_q\} \tag{7.2}$$

The set P_r of data points within a region r and the set Q_r of queries which intersect region r are defined in a similar fashion.

$$P_r = \{p \in P \mid p \in A_r\} \tag{7.3}$$
$$Q_r = \{q \in Q \mid A_q \cap A_r \neq \emptyset\} \tag{7.4}$$

7.2.1 Point Weight

If a partitioning scheme is targeted at balancing data skew, the weight of a data point is relevant. Data skew can originate from data spaces with a mix of densely and sparsely populated regions. The differences in data density may arise from the original data distribution or from the fact that some regions have been investigated more extensively than others, i. e., more data has been collected and is available.

In general, we can define the weight $\mathrm{w}(p)$ of a point p as a function of its default weight σ and the queries Q_p it is relevant for:

$$\mathrm{w}(p) = \sigma + f(Q_p) \tag{7.5}$$

When weighting data points, each point has a default weight σ, e. g., $\sigma = 1$. Now, we also want to consider queries for which a point p is relevant. For example, if a point is relevant for 10 queries, it will have an additional weight of 10. Note that if we set default weight $\sigma = 0$, only data points which are relevant to *any* query are considered during the training phase.

Example 1: Cardinality Function In the introductory example from above, we used the function $f : Q_p \mapsto |Q_p|$. It is a reasonable candidate function: easy to understand, strictly monotonically increasing, and easy to compute.

Example 2: Scaled Weight Function for Point Data The extent of the actual query hot spot(s) is unknown during the training phase. While we can locate the positions of query hot spots with our representative training workload Q, we can only approximate the extent of the area of the data space that will be subject to high query workload. Cases where only a limited number of queries is available during training make such estimates more difficult. As a consequence, we try to approximate the actual hot spots by increasing the query area A_q for all queries q by a *scaling factor* $\phi \geq 1$ in every dimension. We denote this area as $A_{q,\phi}$ in the following.[1] Also, we introduce a new parameter λ in the weight function of data points in order to scale the importance of Q_p in relation to the default weight σ. Thus, we extend Equation 7.5 to:

$$\mathrm{w}_{\text{scaled},\lambda,\phi}(p) \quad = \quad \sigma + \lambda \cdot |\{q \in Q \mid p \in A_{q,\phi}\}| \tag{7.6}$$

It is important to note that tuning the parameters ϕ and λ for $\mathrm{w}_{\text{scaled}}$ can be quite difficult. Choosing the wrong scaling factor can yield counterproductive partitioning schemes, which was confirmed by our experimental results (Section 7.3.1).

7.2.2 Query Weight

The weight for queries is defined in a similar fashion. We assign a default weight γ to each query, which represents the default processing cost for any query. Depending on the set P_q, we add an additional query weight $g(P_q)$. In the following, we use $g(P_q) = |P_q|$.

$$\mathrm{w}(q) \quad = \quad \gamma + g(P_q) \tag{7.7}$$

7.2.3 Combining Data and Query Weights

The weight functions for data points (Equations 7.5 and 7.6) and queries (Equation 7.7) constitute the basic building blocks for defining the weight of a region. Based on the weight of the individual partitions, we always split the partition with the highest weight next. The weight of a region r depends on a function h of the data points P_r it contains and a function i of the queries Q_r which intersect with its area.

$$\mathrm{w}(r) \quad = \quad h(P_r) \otimes i(Q_r), \quad \text{where} \otimes \in \{+, \cdot\} \tag{7.8}$$

In the following, we will only discuss the multiplication of both weights ($\cdot$ is used for $\otimes$).

Combining the weight functions for points and queries to the weight of a region will result in partitioning schemes that are optimized for various load balancing goals. Depending on the combination, a partitioning scheme can achieve load balancing for data, for queries, or for both. In Table 7.2, we summarize five general patterns of how the weight functions of points and queries are combined for the weight of a region. We associate the examples discussed in the following with their corresponding pattern and state their load balancing capabilities.

Weight functions for the first two approaches consider either only the data points or only the queries for computing the weight of a region. The first (w_p) just counts the data points within a region, the second (w_q) only considers the number of queries intersecting with each partition.

[1] Equation 7.1 and 7.2 are still valid, as $A_{q,1} = A_q$.

Points	Queries	Regions	Example	Load balancing
1	–	$h(P_r)$	w_p	Data
–	1	$i(Q_r)$	w_q	Queries
$f(Q_p)$	–	$h(P_r)$	w_{Q_p}	Data and queries
–	$g(P_q)$	$i(Q_r)$	w_{P_q}	Data and queries
1	1	$h(P_r) \cdot i(Q_r)$	w_{pq}	Data and queries

Table 7.2: Overview of region weight functions in Section 7.2.3

All remaining three alternatives consider both data and queries for weighting the regions. In the third and fourth approach (w_{Q_p} and w_{P_q}), the weight of a data region only depends on one of the building blocks—either points or queries—however the weight of the chosen building block is influenced by the other, e. g., we weight each data point according to its relevance for queries. The last weight function w_{pq} computes the *heat* of a region by multiplying the number of objects within a region and the number of queries intersecting the region. This weight function implicitly scales all queries until they cover the entire area of the region(s) they intersect and assigns more weight to regions that contain lots of data and receive many queries. Thus, this weight function provides a notion of the overall load based both on data and on queries. If the region contains no data, its weight is 0 and therefore it is unlikely to be split. If regions receive no queries, we prefer to split those regions that contain more data. We therefore use $(|Q_r|+1)$ in case of queries. Thus, if a region receives no queries, it still has the same weight as when using w_p.

After we have introduced the five weight functions intuitively, the Equations 7.9–7.13 give their formal definition.

$$\mathrm{w}_p(r) = |P_r| \tag{7.9}$$

$$\mathrm{w}_q(r) = |Q_r| \tag{7.10}$$

$$\mathrm{w}_{Q_p}(r) = \sum_{p \in P_r} \mathrm{w}(p) \tag{7.11}$$

$$\mathrm{w}_{P_q}(r) = \sum_{q \in Q_r} \mathrm{w}(q) \tag{7.12}$$

$$\mathrm{w}_{pq}(r) = |P_r| \cdot (|Q_r|+1) \tag{7.13}$$

The relevance of data points for a particular query q can also be described using the indicator function $\mathbf{1}_{A_q} : P \rightarrow \{0,1\}$:

$$\mathbf{1}_{A_q}(p) = \begin{cases} 0 & \text{if } p \notin A_q, \\ 1 & \text{if } p \in A_q. \end{cases} \tag{7.14}$$

Equation 7.15 shows that w_{Q_p} and w_{P_q} define the same weight function if we set the default weights to $\sigma = \gamma = 0$ in Equations 7.5 and 7.7 and $f(Q_p) = |Q_p|$ and $g(P_q) = |P_q|$, respectively. Intuitively, counting points weighted by the queries they are relevant for is equivalent to counting queries weighted by the points that are relevant for them.

$$\mathrm{w}_{Q_p}(r) = \sum_{p \in P_r} \mathrm{w}(p) = \sum_{p \in P_r} \sum_{q \in Q_r} \mathbf{1}_{A_q}(p) = \sum_{q \in Q_r} \mathrm{w}(q) = \mathrm{w}_{P_q}(r) \tag{7.15}$$

For the sake of simplicity, we use the weight-factors σ, γ, ϕ, and λ as constants. Other scenarios, where σ, for example, is a function that returns the average size of a data point p depending on the catalog it originates from, are also applicable but are not discussed further.

7.2.4 Adding Query Extents

The pure heat-based weight function w_{pq} captivates with its simplicity. However, if there is a small hot-spot area, heat-based partitioning may split that area multiple times as it tries to reduce the query load imbalance. This can lead to communication-thrashing, i. e., too much communication between nodes covering neighboring partitions to retrieve the complete result.

For example in our two-dimensional quadtree-based partitioning schemes for astrophysics data, a query area A_q containing the centroid of the region area A_r, would be split into four subqueries. In the worst case, four different nodes are responsible for these regions. This would result in four-times overhead, as intermediate results need to be transmitted and the query uses CPU resources on four nodes. Under such circumstances, we prefer to keep the region as a whole and rather replicate it with our master-slave approach as described in Chapter 8.

Our replication-aware weight function w_{A_q} incorporates the extents of queries and regions by classifying the queries according to the fraction of the area A_q of query q and the area A_r of region r. Thus, the weight function w_{A_q} realizes the behavior from Table 7.1 in Section 7.1. For $0 < \alpha \leq \beta < 1$, the sets of small (big) queries Q_r^{small} (Q_r^{big}) are defined in Equations 7.16 and 7.17, respectively.

$$
\begin{aligned}
Q_r^{small} &= \{q \in Q_r \mid A_q \leq \alpha \cdot A_r\} \qquad (7.16)\\
Q_r^{big} &= \{q \in Q_r \mid A_q > \beta \cdot A_r\} \qquad (7.17)
\end{aligned}
$$

Based on the classification for Q_r, we define the *splitting gain* for region r, $\mathrm{gain_s}(r)$, as the number of small queries in r (Equation 7.18). Analogously, we define $\mathrm{gain_r}(r)$, the *replication gain* for region r with the number of big queries intersecting r (Equation 7.19). For the same reason as in w_{pq}, we add one to both cardinalities to deal with regions that receive no queries or whose query sets Q^{small} or Q^{big} are empty.

$$
\begin{aligned}
\mathrm{gain_s}(r) &= |Q_r^{small}| + 1 \qquad (7.18)\\
\mathrm{gain_r}(r) &= |Q_r^{big}| + 1 \qquad (7.19)
\end{aligned}
$$

The replication-aware cost function w_{A_q} compares $\mathrm{gain_s}(r)$ and $\mathrm{gain_r}(r)$ to determine whether a region should be split or not. As long as splitting a region is considered beneficial, only the value of $\mathrm{gain_s}(r)$ is used. As soon as the "big" queries outnumber the "small" queries, we reduce the weight of a region considerably, by multiplying the size of $|P_r|$ with the fraction of the small queries and big queries. In some application domains it might be desirable to additionally specify the preference τ for either splitting or replication. This is formalized by Equation 7.20.

$$
\mathrm{w}_{A_q,\alpha,\beta,\tau}(r) = \begin{cases} |P_r| \cdot \dfrac{\mathrm{gain_s}(r)}{\mathrm{gain_r}(r)} & \text{, if } \tau \cdot \mathrm{gain_s}(r) < \mathrm{gain_r}(r)\,, \\ |P_r| \cdot \mathrm{gain_s}(r) & \text{, otherwise}\,. \end{cases} \qquad (7.20)
$$

The values for α and β strongly depend on workload characteristics of the application domain, as we realized during our evaluation. At first thought, values like $\alpha = \frac{1}{4}, \beta = \frac{3}{4}$ or $\alpha = \frac{1}{10}, \beta = \frac{9}{10}$ seem reasonable. Remarkably, those combinations have a fairly large "blind angle", i. e., they ignore queries which have area extents between both thresholds. Especially, the decision in favor of splitting a region is sensitive to this gap. After having made the decision to split a region, those "hidden" queries will probably intersect multiple regions causing high overhead. Thus, we suggest to use the same values for α and β.

7.2.5 Cost Analysis

The complexity of the weight functions described in the previous sections strongly depends on the choice of functions f, g, h, and i as well as on the data structures used for storing the training set P and the training workload Q. A naive approach iterating over all queries in the workload in order to acquire the weight for all data points would lead to an overall complexity of $O(|P| \cdot |Q|)$.

The complexity and overhead of maintaining the data points and queries as well as the complexity of performing the weighting can be reduced via appropriate data structures. We use hierarchical, tree-like data structures, e. g., quadtrees (Finkel and Bentley, 1974; Samet, 1990) for creating our partitioning schemes and for storing data points and queries. The leaves of the quadtree correspond to the individual regions. Trees offer a good pruning capability, i. e., one can decide quickly whether a point or a query is relevant. We store both, queries and points, in the same index structure. This allows us to reduce both, the number of data points and queries which need to be considered for computing the weight of a region.

We decided to redundantly store queries which span multiple regions at the leaf-level, i. e., at every region, instead of storing them at inner nodes of the tree, e. g., the nodes that fully contain the bounding box of the hyperrectangle. This further simplifies computing Q_p and Q_r because we do not inspect query sets at inner nodes on the path from the root to the leaf-level.[1] For computing weight functions such as w_{Q_p} or w_{P_q}, containment queries are necessary to decide which query areas contain a data point. These queries can become quite complex, especially if large query workloads are used. Computing the heat of a region by using the weight function $\mathrm{w}_{pq}(r)$, however, is compellingly simple. We only need to multiply the sizes of the two sets P_r and Q_r in order to compute the weight of a region and avoid the cost for comparing each data point of P_r with each query in Q_r. These adaptions further reduce the complexity to compute the weight of a region to $O(|P| + |Q|)$. Only when a region is split, we need to reorganize the sets P_r and Q_r. To summarize, by using a hierarchical data structure for creating the data partitions, we integrate most of the weighting cost into the tree maintenance and only need two lookups in order to compute w_{pq}. For the replication-aware weight function w_{A_q}, we also use the maintenance methods of the tree. When splitting a region, its queries are immediately classified for the newly created leaves into the corresponding sets Q^{small} and Q^{big} by at most two comparisons. Thus, only two additional counters for storing the values of $\mathrm{gain_s}(r)$ and $\mathrm{gain_r}(r)$ are necessary.

7.3 Evaluation

In the following, we will present our analysis of the various workload-aware partitioning schemes introduced in the previous section. Two aspects were important for our evaluation settings: the analytical properties of the partitioning schemes and their impact on the overall throughput in both a simulated and a real deployment. In our opinion, complementing results obtained from statistical analysis and simulations with experiments of an actually deployed system is fundamental for assessing distributed architectures.

Parameter	Value(s)	Description
P	P_{obs}, P_{mil}	Data sets used for training sample extraction
Q	Q_{obs}, Q_{mil}	Workloads used for workload-aware training
s	0.1%, 1%, 10%	Size of the training set
n	4^2, 4^3, 4^4, 4^5, 4^6, 4^7, 4^8, 4^9	Size of the partitioning scheme

Table 7.3: General parameters for the evaluation of workload-aware partitioning schemes

Weight function	Parameter and value(s)
w_{Q_p}, $\mathrm{w}_{\mathrm{scaled},\lambda,\phi}$	$\lambda = 0.01$, $\phi = 10, 20, 40, 80$ (for P_{obs}) $\lambda = 1$, $\phi = 10, 50, 100, 200, 400$ (for P_{mil}) $\sigma = 1$ (both data sets)
w_{A_q}	$\alpha, \beta \in \{ \frac{1}{4096} (\approx 0.0002), \frac{1}{256} (\approx 0.004), \frac{1}{16} (= 0.0625), \frac{1}{4} (= 0.25) \}$, $\tau = 1$

Table 7.4: Weight function specific parameters

7.3.1 Partitioning Scheme Properties

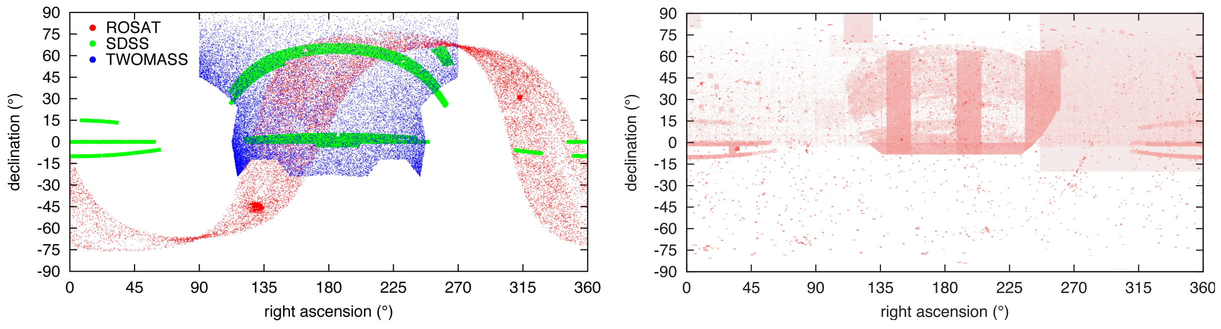
In order to evaluate the analytical properties of the partitioning schemes created with the *data-based* (w_p), *query-based* (w_{Q_p}), *heat-based* (w_{pq}), and *extent-based* (w_{A_q}) weight functions outlined in this chapter, we conducted several experiments. We give a detailed description of the parameters, data sets, and query workloads used during our evaluation. We present and discuss results with respect to our goal of achieving a workload-aware data partitioning.

For the evaluation, we used our Java-based prototype HiSbase which created partitioning schemes according to our weight functions, based on a training data sample P and a representative query workload Q. The remaining queries, not used during training, were used during testing and evaluating the partitioning schemes. We applied a tenfold cross-validation, which is a common setup for machine learning techniques (Witten and Frank, 2005). First, we evaluated the performance of our technique on data samples from three astrophysical catalogs using a query trace from the SDSS catalog. Afterwards, we assessed the effect of our workload-aware training on a data sample from astrophysical simulations using a synthetic workload. This gave valuable additional information, as the simulation data is quite uniformly distributed and so the impact of some parameters was more clearly visible.

We constructed quadtree-based partitioning schemes using the standard splitting strategy as well as the median-based heuristics from Chapter 3. Here, we only discuss results with standard quadtrees. For each approach, we varied the number of partitions to be all powers of four between 16 (4^2) and 262 144 (4^9) as these can be generated exactly by quadtrees and correspond to the values used in the previous evaluations.

For both data sets used during the evaluation, we drew several training samples of different sizes (0.1%, 1%, and 10%) to benchmark the quality of results produced from small data sets. We extracted the random samples with functionality provided by relational database systems. We report on the results obtained from quadtree-based partitioning schemes using the standard splitting strategy based on the 0.1% and 1% samples, each containing about 150 000 and

[1]This is basically the same trade-off as between MX-CIF and extended MX-CIF quadtrees (Samet, 1998).



(a) Data set P_{obs} **(b)** Query set Q_{obs}

Figure 7.4: The observational data and workload

1 500 000 data points, respectively. Table 7.3 summarizes the general parameters used during the evaluation.

From the weight functions defined in Section 7.2, we used the uniform point (*data-based*) weight function w_p as a baseline for our comparisons, and the *query-based* point weight function w_{Q_p} with various values for the default weight σ, the importance λ of Q_p, and the scaling factor ϕ. Furthermore, we used the *heat-based* weight function w_{pq} and the query *extent-based* weight function w_{A_q} with $\tau = 1$ and with varying α and β thresholds. Table 7.4 gives a summary of the used weight function parameters.

Results from the Observational Data Set

We first report on the results from the observational data set P_{obs} and its accompanying query workload Q_{obs} as described in Section 1.2 (both shown in Figure 7.4, repeated for convenience).

Parameters for Extent-based Partitioning Schemes The values for the parameters of w_{A_q} are motivated by the observation in Section 7.1 that the fraction between a query and a region increases four-fold with every split. Analyzing the workload Q_{obs}, we found that the median of the used search radii in Q_{obs} is at 0.2 arc minutes and 75% of the queries have a radius smaller than 0.4 arc minutes. This is extremely small, as queries with an 0.2 arc minutes radius cover only $\frac{1}{10^9}$ of the whole sky. If the queries are at such a small scale, we need to adapt β accordingly. For example, our smallest value of β, 0.0002, corresponds to $\frac{1}{4^6}$ and classifies those queries as big that will increase the network load with high probability, when their region is split an additional six times.[1]

Spatial Locality and Small Partitioning Schemes During the course of our evaluation, we made several observations. First of all, the example workload Q_{obs} from the SDSS archive, shows the expected high spatial locality, as can be seen from Figure 7.5. With all tested partitioning schemes, for less than 10% of the queries multiple partitions contain relevant information. Communities with such workload characteristics greatly benefit from the high degree of parallelism within the system. For up to 1 024 histogram regions, the number of one-region queries is identical and the partitioning schemes only have minor differences. Therefore, partitioning data—even with a partitioning scheme which is only based on the data—can migrate

[1]Each split in quadtree replaces one leave by four new leaves. Thus six splits create 4^6 leaves.

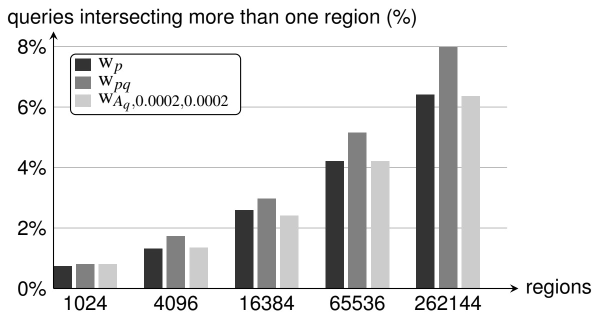
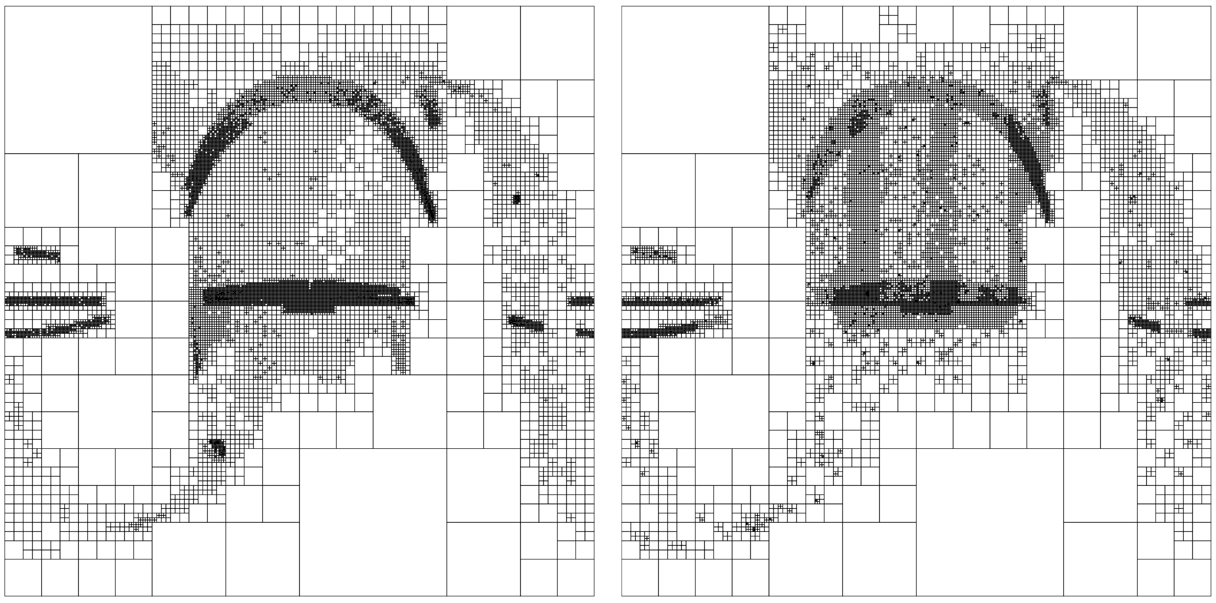


Figure 7.5: Percentage of queries in Q_{obs} that are answered by consulting more than one partition



(a) w_p **(b)** w_{pq}

Figure 7.6: Quadtree-based partitioning schemes of P_{obs} with 16 384 regions

load to different partitions which work in parallel. For uniform query loads and communities at the very beginning of building their grid infrastructure, data-based partitioning is completely sufficient.

Adaption to Query Workloads When we compare the partitioning schemes of P_{obs} in Figure 7.6 with the original data and query set from Figure 7.4, we can observe the similarity between the data distribution and the data-based weight function w_p and also the heat-based partitioning w_{pq} and the query load Q_{obs}. Thus, we can see that our weight functions are able to create workload-aware data partitionings.

Load Balancing Capabilities Figure 7.7 shows that the heat-based weight function w_{pq} distributes the overall load significantly better across multiple nodes for quadtree-based partitioning schemes on the observational data set using the SDSS workload than the weight function w_p which focuses on data load only. We quantitatively evaluated the total query load by calculating the sum of the individual query loads for each region, and the uniformness of the load

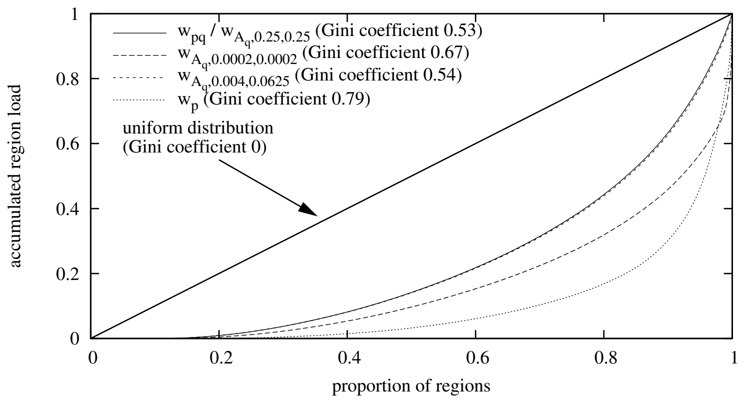


Figure 7.7: Lorenz curves for P_{obs} for partitioning schemes with 4 096 regions and weight functions w_p and w_{pq}

distribution using the Gini coefficient as in (Pitoura et al., 2006). The Gini coefficient is defined as the area between the Lorenz curve for the distribution and the diagonal.

While w_p only achieves a Gini coefficient of 0.79, w_{pq} has a coefficient of 0.53, which is considered a fair load distribution in distributed systems (Pitoura et al., 2006). The partitioning scheme of $w_{A_q,0.25,0.25}$ achieves the same load distribution and $w_{A_q,0.004,0.0625}$ differs only marginally from the heat-based partitioning. With 0.67, the Gini coefficient of $w_{A_q,0.0002,0.0002}$ lies between w_{pq} and w_p.

In the w_p approach for data load balancing, 20% of the regions receive 83% of the overall system load. For our workload-aware technique w_{pq}, these 20% handle less than 60% of the overall load. When using the extent-based $w_{A_q,0.0002,0.0002}$, 20% of the regions process 68% of the load, as the weight function recognizes some candidate regions for replication. Thus, the query load is less balanced as in the w_{pq} partitioning scheme. We will see in the following that the extent-based approach is preferable to the heat-based technique with regards to other characteristics.

Stable Results with Varying Training Workload Size Furthermore, we investigated how the size of our training workload influenced the results. We used 1%, 5%, 10%, and up to 90% of the queries for training and the remainder for testing. Figure 7.8 shows the results for the w_{pq} weight function. We see that for partitioning schemes having between 1 024 and 16 384 partitions w_{pq} not only achieves a good load balancing, i. e., Gini coefficients close to 0.5, but also that most measurements are independent of the size of the training workload. This stability is an important characteristic as it allows communities to find the trade-offs for using as much queries as possible during the training phase or keeping enough queries to validate the created partitioning scheme.

Regions without Queries We furthermore analyzed, how many regions do not take part in query processing depending on the weight function. The less such regions, the better the weight function distributes the load to several partitions. w_{pq} always achieves the best result. Up to 65 536 regions, both w_{pq} and w_{A_q} are at a comparable level and between 15% and 50% better than pure data load balancing. In Figure 7.9, w_{pq} always has the lowest number of regions without queries. The following analysis, however, shows that w_{pq} is too eager in splitting regions further and further and therefore introduces significant communication overhead.

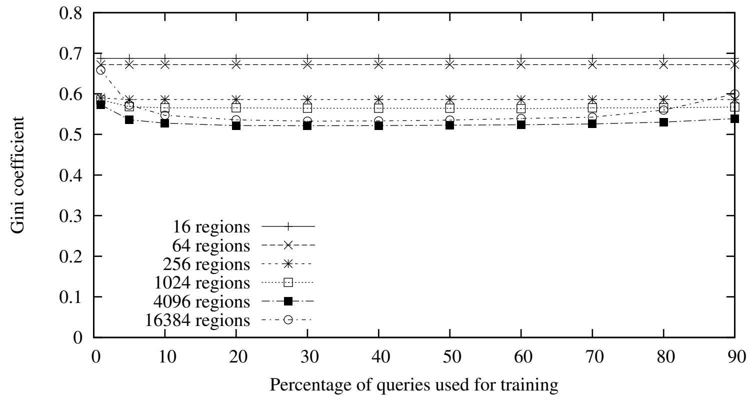


Figure 7.8: Comparison of the Gini coefficients for different training workload sizes for Q_{obs}

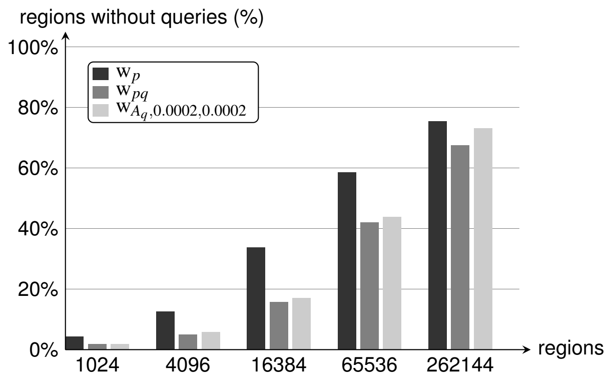


Figure 7.9: Comparison of the percentage of regions that receive no queries from Q_{obs}

Reduced Traffic by Workload-Awareness In order to investigate the communication overhead, we compared our partitioning schemes to a scheme where every query could be answered by a single region. All regions that need to be contacted additionally, increase the communication overhead of the specific weight function. Based on our observation that typical workloads exhibit a high degree of spatial locality, we prefer queries that intersect a single region which is guaranteed to be mapped to one peer. Formally, we compute the communication overhead as

$$\frac{\sum_{q \in Q} |\{r \in R \mid A_q \cap A_r \neq \emptyset\}|}{|Q|} \tag{7.21}$$

Figure 7.10 shows the relative traffic overhead for w_p, w_{pq}, and some variations of w_{A_q}. We clearly see that w_{A_q} produces lower traffic than w_{pq}. Moreover, with a reasonable choice of α and β, w_{A_q} produces not more traffic than w_p.

Results from the Millennium Data Set

Finally, we shortly discuss how the different weight functions performed on the uniform *Millennium* data set P_{mil} (Figure 1.4) with the query workload Q_{mil}.

The query areas for Q_{mil} were artificially generated with their midpoints (p_x, p_y) following a two-dimensional Gaussian distribution with mean $(0,0)$ and variance chosen in such a way that 90% of the midpoints fall into the square area in the center taking 10% of the space. The actual query areas were then constructed around the midpoints from $(p_x - r, p_y - r)$ to $(p_x + r, p_y + r)$

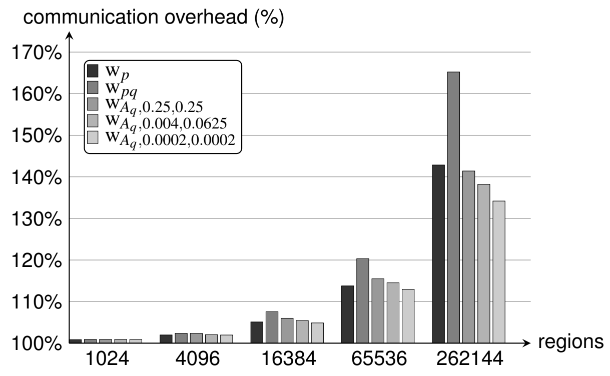


Figure 7.10: Communication overhead for partitioning schemes of P_{obs} in comparison to a centralized setting

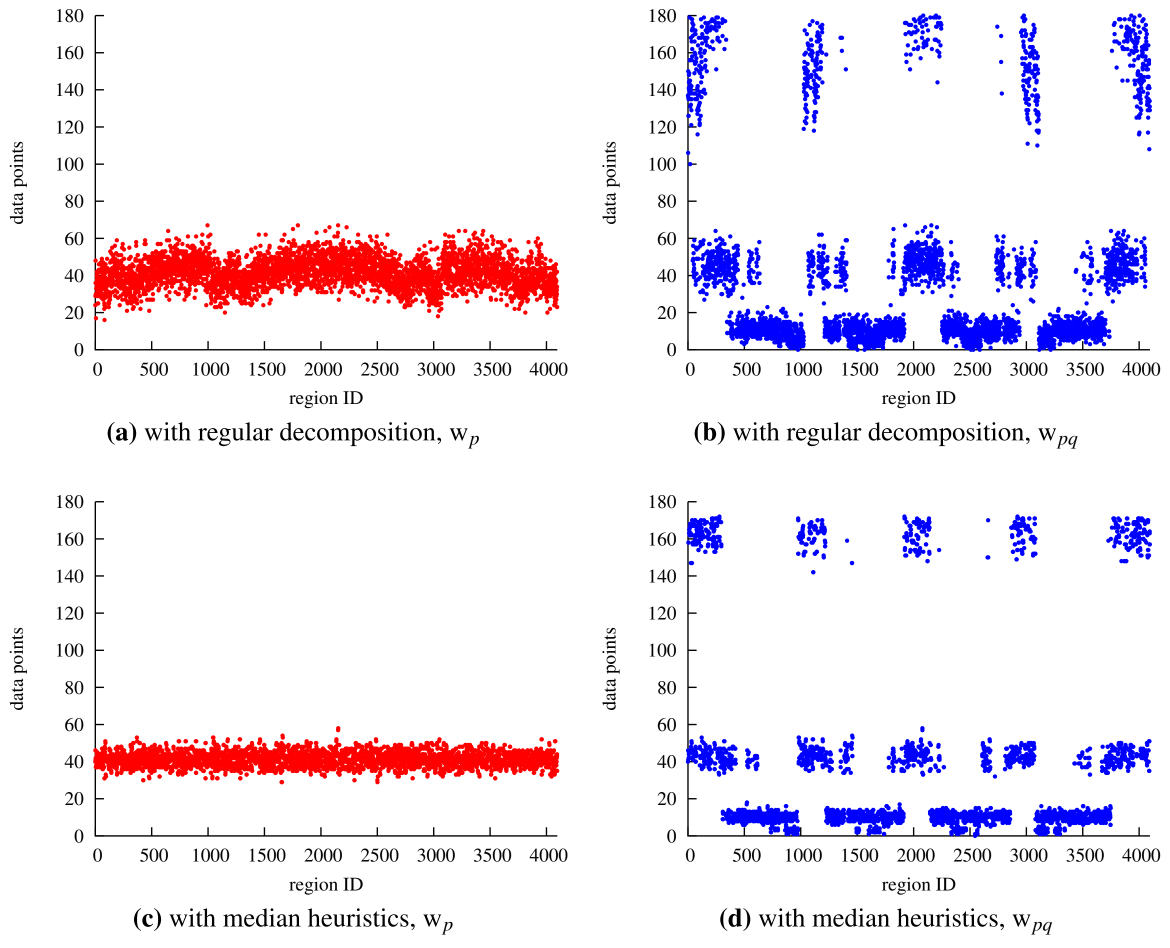
with the query "radius" r chosen randomly from $\{0.025, 0.1, 0.2, 0.25, 0.5\}$ arc minutes, which correspond to the 5 most frequent query radii from the query workload Q_{obs}, introduced above. In this way, 11 000 queries were generated for training and testing the resulting partitioning schemes.

When comparing the data load on each partition (Figure 7.11), the data-based weight function (w_p) achieves, as expected, a good data load balancing for uniform data sets (Figure 7.11(a)). Each quadtree-based partition contains about 40 data objects from the 0.1% training sample. Using the median-heuristics further reduces the variation in the amount of data, shown in Figure 7.11(c) by the "tighter" band of region sizes. For the workload-aware partitioning (w_{pq}), data distribution is less uniform (Figure 7.11(b)) as the size of the partitions is adapted to the workload. Especially when looking at the median-based partitioning (Figure 7.11(d)), four groups of partitions become evident: 4-times larger (about 160 objects), unchanged (40 objects), 4-times smaller (10 objects), and 16-times smaller (about 3 objects).

In our motivation, we suggested to use the amount of data inside a region as indicator for whether a region is hot. When looking at the combined load of each region in Figure 7.12, we see four "flames" of high heat[1] for the quadtree-based partitioning schemes with regular decomposition for both weight functions w_p (Figure 7.12(a)) and w_{pq} (Figure 7.12(b)). When compared to the data-based partitioning (Figure 7.12(a)), however, the heat levels are greatly reduced when using the heat-based weight function. The results for the splitting strategy based on our median heuristics (Figures 7.12(c) and (d)) are very similar in Figure 7.12. However, we show both to demonstrate how a *low data load* for the heat-based partitioning (Figure 7.11(b) and (d)) correlates with a *high region load* for the data-based weighting scheme (Figure 7.12(a) and (c)).

In Figure 7.13, some of the partitioning schemes of P_{mil} with 1 024 partitions are shown. Each region is colored with its heat-based weight (the w_{pq}-value) on a scale from cold (white) to hot (red), which is normalized over the compared partitioning schemes. The w_p-partitioning scheme, in Figure 7.13(a), is a completely balanced quadtree with 1 024 same-sized partitions as the data distribution is almost uniform. The hot spot in the center of the data space is also clearly visible as the query load is not considered. Figure 7.13(b) shows how w_{Q_p} splits the hot regions first and the partitioning scheme adapts to the hot spot. As the number of regions is fixed, regions at the border of the data space are not split further and now contain more data. From Figure 7.13(c) it becomes obvious why: regions at the border of the hot spot contain

[1]We see four "flames" instead of a single "bonfire" due to the Z-order space filling curve.



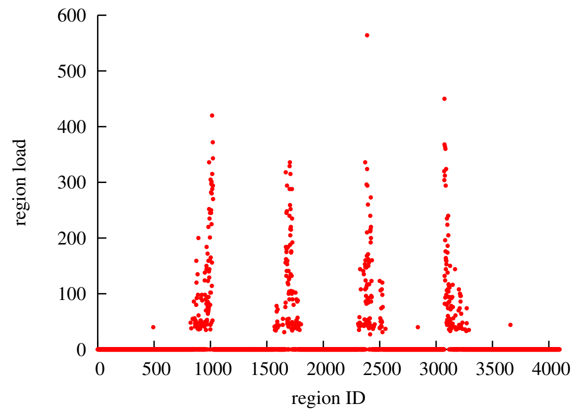
(a) with regular decomposition, w_p

(b) with regular decomposition, w_{pq}

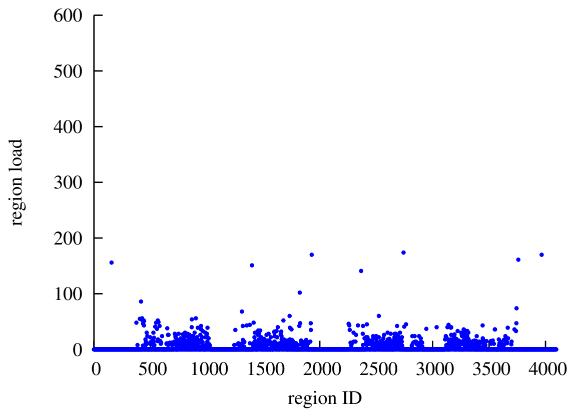
(c) with median heuristics, w_p

(d) with median heuristics, w_{pq}

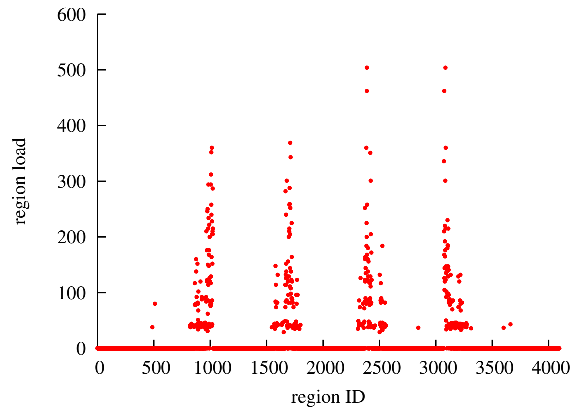
Figure 7.11: Data load of P_{mil} for quadtree-based partitioning schemes with 4 096 partitions for the data-based weight function (w_p) and the heat-based weight function (w_{pq})



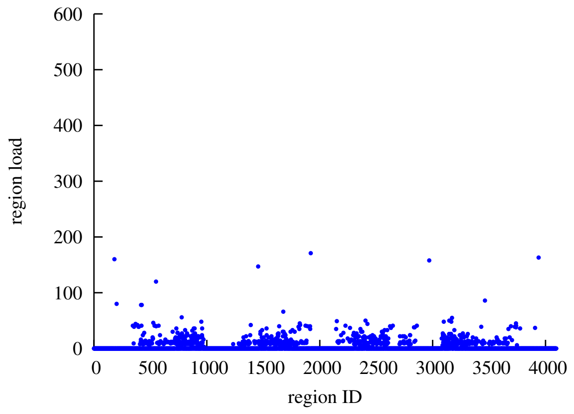
(a) with regular decomposition, w_p



(b) with regular decomposition, w_{pq}

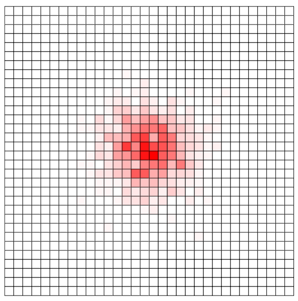


(c) with median heuristics, w_p

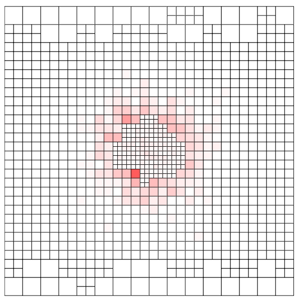


(d) with median heuristics, w_{pq}

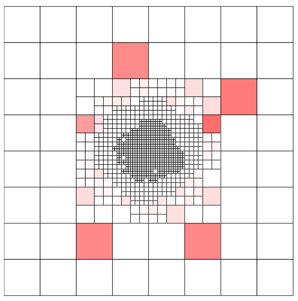
Figure 7.12: Region load of P_{mil} for quadtree-based partitioning schemes with the 4 096 partitions for the data-based weight function (w_p) and the heat-based weight function (w_{pq})



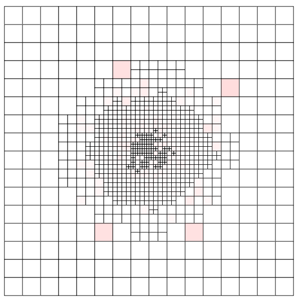
(a) w_p



(b) w_{Q_p}, $w_{\text{scaled},\lambda=1,\phi=50}$



(c) w_{Q_p}, $w_{\text{scaled},\lambda=1,\phi=400}$



(d) w_{pq}, w_{A_q}

Figure 7.13: Partitioning with 1 024 regions for P_{mil}

16-times the data than in the w_p partitioning but also receive many queries and thus their load will be too high. Finally, we see in Figure 7.13(d) that the heat-based weighting scheme w_{pq} approximates the extent of the hot spot very good but does not lose sight of data load balancing. Actually, the extent-based weight function w_{A_q} produced the same partitioning scheme. Note the four different sizes of regions in our workload-aware partitionings as in Figure 7.11(b): some regions are 16-times smaller (in the very center of the hot spot), 4-times smaller, unchanged, and 4-times larger (the cold regions at the border of the data space) than the regions of the w_p partitioning scheme.

7.3.2 Throughput Evaluation

The previous analysis of the histograms was based on the training data and both the training and testing workloads. The following throughput measurements are conducted on distributed or simulated HiSbase instances. We intentionally do not use the master-slave replication approach at runtime as described in Chapter 8 in order to emphasize the throughput variations purely based on the choice of the weight function.

For the throughput experiments we use the same definitions for saturation phase I_{sat} and throughput T as in Chapter 6.2. During the throughput evaluation, we used partitioning schemes created with the *data-based* (w_p), *heat-based* (w_{pq}), and *extent-based* ($w_{A_q,0.0002,0.0002}$) weight functions, respectively. The size of the histogram—4 096 for real and 262 144 for simulated networks—is chosen to be large enough to ensure that all peers are responsible for data partitions. The results shown are the averages built over three evaluation runs in both (the real and simulated) cases.

Each single HiSbase node was configured to allow ten parallel queries on its local database, as suggested in Chapter 6.2. The following evaluation shows that especially higher multi-programming levels for the HiSbase nodes increase the overall throughput; we report on MPLs selected from {10, 50, 100, 300, 500}.

Results from the Observational Workload

For our throughput experiments, we used the same 22 000 queries from Q_{obs} as in the throughput evaluation for coordination strategies in Section 6.2.3. Each node randomly selected 5 000 queries from Q_{eval} for its query batch.

Our 16 computer lab nodes are consumer-class Linux PCs equipped with 1.6 GHz processors, 512 MB RAM and running DB2 V9.1.4. For the measurements with 32 nodes, we additionally used 16 nodes from our local AstroGrid-D resources having between 1 and 4 GB main memory and 2.8 GHz Intel Xeon CPUs. We used the H2 database for performing the local query processing. After assigning each node a random identifier, we distributed the data according to the partitioning scheme.

Figure 7.14 shows the throughput achieved by a HiSbase network using a partitioning scheme with 4 096 partitions which are distributed among 16 and 32 nodes, respectively. For 16 peers, we see that the w_p partitioning scheme surprisingly achieves the highest throughput, while for the setup with 32 nodes, the extent-based technique w_{A_q} outperforms both w_p and w_{pq}. Due to several query hot spots in Q_{eval}, only a fraction of the additional nodes can significantly participate during query processing. While w_{A_q} increases the throughput in a near-linear fashion, the gain of w_p and w_{pq} is only sub-linear.

We have successfully demonstrated HiSbase (Scholl et al., 2007a) on PlanetLab and we see great value of PlanetLab for evaluating the algorithmic properties (like messaging over-

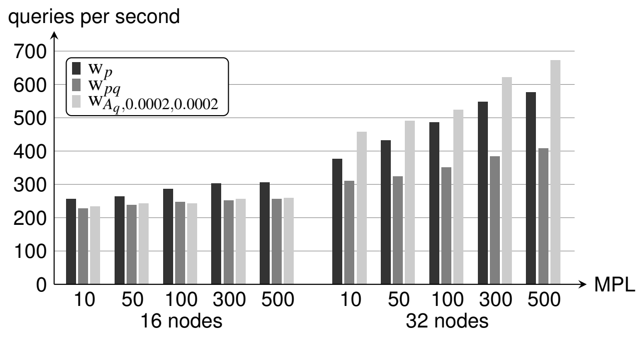


Figure 7.14: Throughput of deployments with 16 and 32 nodes

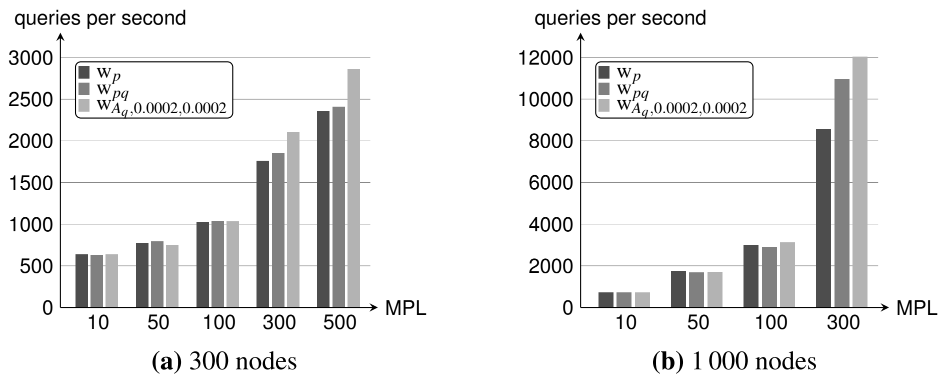


Figure 7.15: Throughput on the observational query workload with simulated networks

head) of distributed architectures in volatile environments. However, our initial throughput measurements on 100 PlanetLab nodes showed that the PlanetLab framework is not suitable for performing throughput evaluations of data-intensive grid applications. Issues like bandwidth-limited links and limited main memory access (below 160 MB) do not reflect the anticipated infrastructure for community grids. We therefore abandoned using PlanetLab and evaluated the throughput trend of larger deployments with FreePastry's discrete-event simulator instead.

During our simulations, we evaluated three different partitioning schemes (w_p, w_{pq}, w_{A_q}) with 262 144 partitions using the MPLs from above on HiSbase networks with 100, 300, and 1 000 nodes, respectively.[1] Database accesses were simulated by returning the result set after a delay extracted from annotations to the queries. The time specified in the annotation corresponded to the running time of the query on the central database server during the selection of the Q_{eval} queries. The simulation engine does not model running time improvements due to caching effects or smaller databases at each nodes. Likewise, we assumed that parallel running queries do not interfere and the annotated running time is also valid for ten parallel queries. Runs with the same setups like in the distributed scenarios with 16 and 32 nodes verified that the results of the simulator are realistic. The ratios between the simulated measurements and the real results were at a reasonable level between 1.07 and 2.25.

The results from the simulations showed a similar trend as in the real deployments. With all tested partitioning strategies, the extreme query hot spots diminished the load balancing effect of adding new nodes with all tested partitioning schemes.

[1]We did not measure the throughput of 1 000 nodes with MPL=500 as current e-science scenarios do not yet require such high parallelism.

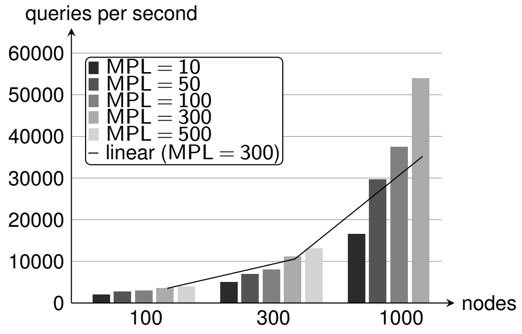


Figure 7.16: Throughput for the region-uniform query workload

Figure 7.15 depicts that workload-aware partitioning schemes perform better for high MPL levels (MPL=300 and MPL=500) and large networks with 300 and 1 000 nodes than the pure data-load approach.

Results from the Region-Uniform Workload

Analyzing the query workload Q_{eval} for the partitioning schemes with 262 144 regions revealed that only 3% of the regions receive any query (as opposed to the 27% in Figure 7.9 based on the complete query set Q_{obs}). In order to evaluate the scalability under a uniform query workload, we generated workloads where 92% of the regions intersect with queries. The 600 000 generated queries were uniformly distributed among the regions and the query areas were constructed in a similar fashion as for the Millennium data set (see Section 7.3.1). According to a uniform distribution, we picked a region identifier and a center point for the query area within that region. As all histograms achieved similiar results, Figure 7.16 depicts only how the extent-based histogram balances the query load uniformly in a near-linear and even super-linear throughput (stressed by the trend line for MPL=300), especially when comparing the 300-node and 1000-node networks.

7.3.3 Summary

In summary, the evaluation corroborates that query hot spots are an important issue in scientific data management. The analytical evaluation showed that our partitioning techniques adapt to the query workload and that our extent-based technique (w_{A_q}) does not raise the message overhead compared to the data-based distribution but it also offers query load balancing. The throughput experiments showed that the extent-based partitioning is best in taking advantage of additional nodes, especially for highly parallel workloads. For all partitioning schemes, throughput significantly improves when all nodes participate during query processing. In the presence of query skew, however, this can only be achieved by replication at runtime. Therefore, employing load balancing at runtime with techniques such as a master-slave approach and replicating "hot" data based on monitoring statistics are the next important steps towards even more workload-aware community-driven data grids and are discussed in the following chapter.

7.4 Related Work

The fact that workloads on astrophysical data sets are mostly spatial queries (selections on the celestial coordinates or corresponding stored procedures) and that these workloads exhibit a high query skew is supported by an extensive analysis of the traffic for the SDSS SkyServer (Singh et al., 2006) and an experience report on migrating the SkyServer on MonetDB (Ivanova et al., 2007).

Papadomanolakis and Ailamaki (2004) propose an automatic categorical data partitioning algorithm based on queries from a representative workload. Categorical partitionings are based on attributes which take only a small number of discrete values and identify objects, i. e., a TYPE-attribute within an astrophysical data set having values like “Star” or “Galaxy”. However, their approach only deals with optimizing the database structure (indices, restructured tables) within the database of a single node.

In the following, we discuss related work (Crainiceanu et al., 2007; Ganesan et al., 2004a; Pitoura et al., 2006) investigating techniques of load balancing for P2P networks. The major difference of these techniques compared to our solution is that load balancing in P2P networks is designed for networks with highly dynamic data and mostly deals with either skewed data distributions or skewed query load, but not both. The flexibility needed in such a fast changing environment comes at the price of dealing with each data object individually, which can result in prohibitive costs for disseminating vast amounts of data to multiple nodes. Ganesan et al. (2004a) show that load balancing schemes for range-partitioned data in highly dynamic P2P networks either need to adjust the load between neighbors or need to change peer positions within the range. HotRod (Pitoura et al., 2006) addresses query hot spots on one-dimensional data by replicating popular data ranges on additional rings but does not deal with skewed data distributions. P-Ring (Crainiceanu et al., 2007) addresses data skew in an orthogonal manner in comparison to the partitioning-based approach, but does not consider query hot spots. While our partitioning schemes adapt the regions to data skew and query skew by distributing these across the cooperating peers, P-Ring has the notion of “helper peers” that support peers which are overloaded by skewed insertions either by data redistribution between neighbors or by merging their data into a neighbor’s range. In P-Ring it is required that there are less data partitions than peers. If P-Ring would be extended in order to support these large data sets by supporting our notion of regions, that requirement would lead to larger partitions which are not as easily distributed as the regions created with our workload-aware partitioning schemes. Furthermore, P-Ring does not perform data replication.

SD-Rtree (du Mouza et al., 2007, 2009) is a *Scalable Distributed Data Structure (SDDS)* which targets large data sets of spatial objects. An SDDS has the following characteristics: 1) It has no central data index, 2) servers are dynamically added to the system when needed, and 3) the clients access the SDSS through an *image* which is potentially outdated. SD-Rtrees perform data load-balancing by data partitioning and reorganization similar to AVL trees. The histogram in HiSbase differs from the SD-Rtree index in that it is used also for query load balancing and it uses the multi-dimensional index structure to determine candidate regions for replication.

Related work in sensor networks (e. g., Aly et al., 2005, 2006) illuminates aspects of data distribution and load balancing from a different perspective where data is created within the network and the predominant goal is to increase quality of data and reduce the power consumption in order to increase the lifetime of a sensor network. As these solutions also deal with individual data objects, it is currently unclear whether they can be directly applied to petabyte-scale data sets of e-science communities. However, it is an interesting question for future investigations.

7.5 Summary and Future Work

Supporting the efforts for building global-scale data management solutions within many e-science communities such as biology or astrophysics is a challenging task. In this chapter, we have described several weight functions to create cost-based partitioning schemes for community-driven data grids that address data skew, query hot spots—each on its own or in combination—and finally, a weight function that only splits data regions if the gain of doing so is higher than the gain of replicating that region.

We evaluated our weight functions on a data set from astrophysical observations and data from an astrophysical simulation with actual application workloads. For small communities, surprisingly simple partitioning schemes already achieved good load balancing results. With increasing number of partitions, the extent-based weight function outperforms the other schemes with regards to reduced communication overhead and load balancing. Based on our throughput evaluation, workload-aware partitioning alone is not sufficient to completely level out query hot spots. As a consequence, we discuss how to incorporate load balancing techniques such as a master-slave hierarchy in our data grid infrastructure in the following chapter.

Further interesting open research issues are adaption to heterogeneous nodes with different capacities or whether the complementary approach of merging *cold* regions can be included in our training phase and how communities can benefit from that.

CHAPTER 8

Load Balancing at Runtime

In the previous chapter, we have provided several weight functions for regions. We apply these region weight functions during data partitioning. Beginning with pure data-based weight functions, we proposed several weight functions that additionally take the workload into account.

In this chapter, we discuss load balancing techniques used in community-driven data grids at runtime. We distinguish between *short-term* imbalance due to sudden and transient changes in the current system workload (Section 8.1) and *long-term* changes in both data and query distributions (Section 8.2).

8.1 Short-term Load Balancing

In order to support overloaded nodes at runtime with additional resources, HiSbase builds a *master-slave* hierarchy: overloaded nodes are *masters* while lightly loaded nodes are *slaves* that store less data or process fewer queries. Slave nodes offer some of their capacity to master nodes in order to achieve load balancing. The master-slave relationship is defined with regard to a single region. Thus, a node can be master for one of its own regions as well as slave for other regions in parallel. Once connected with a slave node, master nodes send some of their frequently accessed—so-called *hot*—data subsets to this node. During query processing all replicas are available for query load-balancing purposes.

If a region is replicated on another node, this information is stored in a *replica dictionary* at the master node. The dictionary uses region identifiers as keys and stores lists of nodes as values. For a particular region identifier the list contains the nodes with a copy of the region.

8.1.1 Replication Priority

Having distributed the partitions according to the partitioning scheme, we need to ensure at runtime that we replicate hot regions more often than regions that are likely to be relevant for only a few queries. By defining a *replication priority* for each region, we provide a notion for the urgency for copying this region to a different node. Regions having a high replication priority need to be replicated more often.

Initially, the following priorities can be computed based on the information from the training phase and define a starting point for a static replication. Combined with load monitoring, these priorities can also be dynamically maintained and thus result in a dynamic data replication.

Training-Based Replication Priority $\varphi_{training}$ During the training process, we can identify the replication candidates by our replication-aware weight-function w_{A_q} as defined in Equation (7.20) on page 89. At the end of the training phase, we annotate those regions accordingly. Nodes then can explicitly prefer those regions for replication. Besides annotating hot regions we can furthermore annotate empty regions, which received no data during training. Thus, we can exclude empty partitions from the replication process.

Size-Based Replication Priority φ_{size} For defining the replication priority φ_{size}, we reconsider the heat-based weight-function w_{pq} (from Chapter 7.2.3). The weight function defines the weight of a region as the product of the number of objects and the number of queries within that region. Let P and Q be the set of all data points and queries, respectively. Let P_r and Q_r denote the corresponding subsets that reside within region r. Let w be the weight for any region, then

$$\begin{aligned} \mathrm{w} &= 1 \\ \mathrm{w} &= |P| \cdot |Q| \\ |\mathrm{P}| &= \frac{1}{|Q|} \end{aligned} \tag{8.1}$$

From Equation (8.1) it becomes evident, that w_{pq} focuses on reducing the overall load for queries and data on individual regions. The equation states that the number of data objects in a region is inversely proportional to the number of queries on this region, assuming the load is evenly distributed over all regions r. The fewer data objects are within a region the more queries this region receives. In other words: the real amount of data in these regions will be less due to the increased weight from the queries.

Using R_r as the *replica set* of a region r, i. e., the set of nodes that maintain a copy of region r, we define the size-based replication priority φ_{size} as

$$\varphi_{size}(r) = \frac{1}{|P_r| \cdot |R_r|} \tag{8.2}$$

When data set P_r within a region is smaller than a specific threshold, the region is a hot spot with high probability and needs to be replicated. A good estimate for such a threshold could be the overall data volume divided by the number n of regions in our partitioning scheme: $\lceil |P|/n \rceil \cdot k$, given a node covers k regions. We note that this priority assumes the validity of Equation (8.1), i. e., the overall load is evenly distributed. Otherwise, the replication priority does not offer reliable results.

Query-Based Replication Priority φ_{query} With φ_{query}, we describe the “pressure” on a region that grows with increasing number of queries and decreases with additional replicas.

$$\varphi_{query}(r) = \frac{|Q_r|}{|R_r|} \tag{8.3}$$

For a community-driven data grid with similar resources, we use a global threshold for the query-based replication priority. However, finding a suitable threshold for the query-based replication priority is more difficult than for the size-based priority. For heterogeneous environments, probably more advanced approaches are required. For example, each node can exchange replication priorities with its neighbors in order to assess which regions are overloaded.

The query-based replication priority φ_{query} can even detect new short-term query hot spots when we combine it with monitoring statistics about queries and the current load on a node.

8.1.2 Monitoring Statistics

In order to monitor the load on each node, we need to store the statistics about the queries currently running on a node, including their query text and the extracted query area. On a node, queries can have one of three states: waiting, active, or processed. These states represent whether a query is currently in the processing queue, actively running, or completely processed.

Queries in the waiting-state are migration candidates. Once an overloaded node has replicated some hot spot regions, it can look for waiting queries that can be migrated to replicas instead of being processed by itself.

Once processed, all query statistics are persistently stored, e. g., by writing the query statistics to a log file. Even if the responsibility of a node changes, the log files of processed queries are preserved. An example for such change is if a new node joins the network and overtakes responsibility for a part of regions managed by the existing node.

Region-specific data collected during monitoring comprises concurrently waiting queries per region, the size of a region, and the replication dictionary. These query and region statistics are used to identify the current load of nodes—low load, normal load, or overloaded.

Both monitoring statistics, query log files and region specific data, also deliver important details for reorganizations, e. g., future histogram evolutions.

8.1.3 Master-Slave Replication

Once we identified an overloaded node, we need to decide which data to replicate, how many copies to create, and which HiSbase nodes are suited best for managing the replicas.

For these three criteria we need load statistics as well as information about the stability of the individual data nodes. In the context of community-driven data grids such as HiSbase, load information might be of primary interest as we can assume dedicated, highly available nodes with high-bandwidth network interconnections. Such an assumption seems reasonable when dealing with the anticipated amounts of data. If two machines have the same load profile, i. e., based on the load statistics both nodes are candidates for replication, we prefer to replicate the data of the machine with lower availability.

The number of copies is proportional to the ratio between the load on the overloaded node and the load of neighboring nodes. If the load on a node is a factor of three higher than the load on its neighbors, we need to replicate the data three-times in order to level the load between the individual nodes.

There exist multiple criteria for selecting a node that is responsible for a newly added replica. In general, we can choose between *logical* neighbors (nodes that are neighbors on the underlying P2P key space) and *physical* neighbors (nodes that are close within the physical network).

For the decision, we need to trade off a fast replication that requires high bandwidth, which physical neighbors can provide, against preserving the query locality, which is achieved by replicating data to neighbors on the identifier space. By using logical neighbors, we can preserve query locality and completely move queries spanning multiple regions to a different node.

Within our master-slave approach, we use physical neighbor nodes for replicating regions as these large data transfers are presumably faster within a local area network than transmitting the data over wide area network links. If queries span multiple regions, which are managed by multiple nodes, using physical neighbors further reduces the communication overhead. As there also exist P2P-based multicast implementations, e. g., the spanning tree-based Scribe (Rowstron and Druschel, 2001), HiSbase does not require a centralized component to implement an efficient communication.

With regards to the master-slave communication, nodes have three states based on their current load: helper (low data and query load), normal (regular load), and overloaded (too many queries waiting for being processed). Once a node determines that its current load is below the helper-threshold it subscribes to a multicast "helper channel". Once an overloaded node requires support from a helper node, it sends an anycast on the "helper channel". In contrast to broadcasts, the multicast channel stops transmitting an anycast as soon as a helper node has accepted the request. The helper nodes send their current offers to the requester that decides which are the most suitable helper nodes based on the distance and available resources. The master contacts its preferred candidates and transmits the data to be replicated. Once the replication is successfully completed, the master adds all slaves to the replica dictionary.

During query processing, the master now can balance the load among the several replicas either by a round-robin or more advanced load-based scheme. If a helper node gets overloaded itself, it contacts its master(s) and "quits" jobs in order to lower its own load.

Replicated data can be stored at different places inside the database. One choice is to store replicated data in the same tables as the "own" data of the node. Another choice is to store replicated data in a separate database. When all data is stored in a single table, we need to serialize query precessing as our current query processing would induce too many duplicates. Moreover, queries for data originally covered by the node might be slowed down due to the additional data. We therefore separate the original and replicated data. Using this approach, a node can remove replicated data by deleting the separate database.

8.2 Long-term Load Balancing

With a long-term perspective, our load balancing considers changes in the data and query distribution. At runtime, communities can integrate new data sets (e. g., new catalogs or new versions of existing catalogs) in their data grid as soon as the schema information has been distributed across all nodes. In this case, however, data load balancing is possibly no longer optimal. Having added new data, it is therefore advisable to check whether histogram and data distribution can be further optimized.

While the master-slave approach addresses short-term query hot spots, new and even stronger query hot spots can develop and persist for a long-lasting period. The historical data about processed queries is particularly useful for extracting a shift in interest. We now describe how partitioning schemes can evolve during the lifetime of a scientific federation and adapt to long-term trends in the workload.

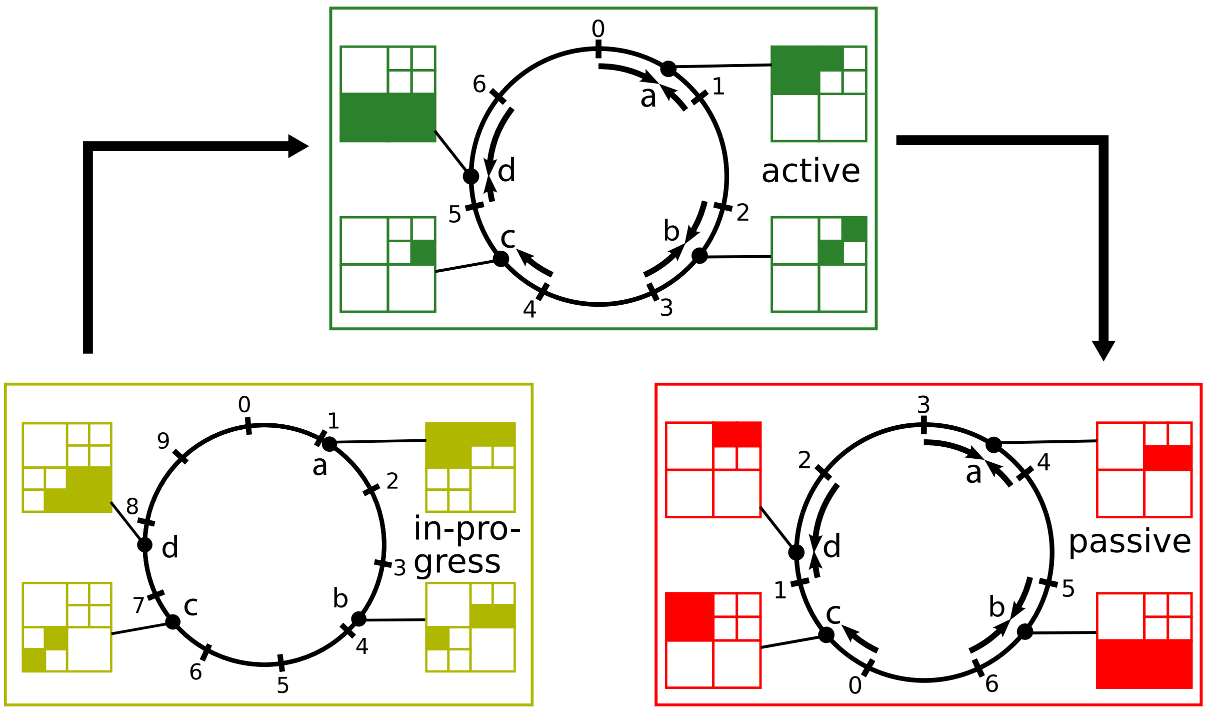


Figure 8.1: Evolution of histograms within HiSbase

8.2.1 Partitioning Scheme Evolution

During the lifetime of a community-driven data grid, additional data sources might be added or the interest of a community might shift towards a different area in the data space. Therefore, we need a mechanism to evolve our histogram. For example, if no previous query logs are available, a community can start with a histogram that is only based on the data distribution. Depending on the change characteristics of the community (e. g., every six months or every year), the data distribution can be reorganized according to the analysis of monitored workloads using a workload-aware partitioning scheme as described in Chapter 7. Therefore, a histogram sometimes needs to be recreated and data needs to be redistributed accordingly.

Each HiSbase histogram passes through three development phases: *in-progress*, *active*, and *passive*. The *in-progress* histogram is used to distribute the data within the HiSbase network. As long as the feeding process (see Section 5) has not finished, histogram and data are not used during query processing. Once the data distribution according to the *in-progress* histogram is completed, the histogram and the newly distributed data are *activated*. Similar to the feeding process, we coordinate the transitions from the different histogram states at the HiSbase nodes using the multicast channel. The active histogram and active data are used during query processing; the histogram for retrieving the relevant data regions and the data for the actual database queries. In order to prolong the use of data once distributed and thus amortize the cost of the data transfers, each node additionally keeps a *passive* histogram and a *passive* data set. The passive histogram is primarily used in order to provide data availability and as fallback support during query processing.

Figure 8.1 shows an example for a snapshot within a histogram evolution. The active histogram has seven partitions and is used during query processing, while the passive histogram functions as backup copy. The region mapping to the key space has a different origin for the passive histogram and therefore is likely to replicate regions to different nodes. The in-progress

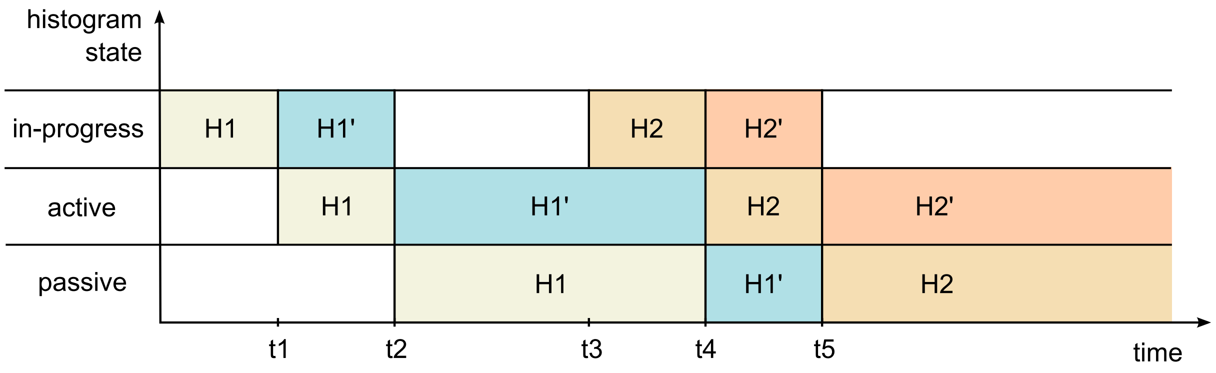


Figure 8.2: Example time line for the transition from histogram H1 to H2. H1′ and H2′ are modified versions the according histogram, e. g., by using a different space filling curve.

histogram has (four) additional regions, e. g., due to a newly added data set (in the lower left corner).

In order to determine the passive histogram and its data set, there are two general design options: 1) use the *outdated* active histogram and its according data and 2) use another slightly modified histogram from the in-progress histogram. Depending on the cause which triggered the histogram evolution—be it a new data source or additional query hot spots—choosing one or the other alternative impacts the system differently.

Given the case that a new data source has been integrated into the HiSbase network together with the in-progress histogram, the outdated (formerly active) histogram does not contain the new data set. As a consequence, the passive histogram can only provide approximate query results for updated data regions. If the new data set has already been integrated into the outdated histogram, the data load balancing according to this histogram might not be optimal—as it triggered the creation and dissemination of a new histogram—but at least the complete data is accessible via the outdated histogram. Whenever a new query hot spot triggers the next histogram evolution, the outdated histogram does not optimally balance query processing load but it contains all data.

Opposed to option 1), the second alternative builds the passive copy already based on the in-progress histogram. We use the same partitioning scheme twice and distribute the data twofold. By rotating the origin of the linearization function (as in Figure 8.1) or by using a different space filling curve we achieve a twofold data availability. This alternative is applicable for the initial setup of a HiSbase instance, i. e., if no new data and statistics are available. Distributing the data for the second copy can be accelerated by our feeding technique for replication as described in Chapter 5. The decision which copy to use can be based on the monitored load information or on a round-robin scheme similar to the master-slave replication.

8.2.2 Data Dissemination during Histogram Evolution

In the following, we illustrate the data distribution during histogram evolution. Figure 8.2 depicts a time line that shows the transition for the histograms H1 and H2. At time t_1, the histogram H1 is activated and we use this histogram together with its data for query processing. In parallel, we replicate the data of H1 according to a slightly modified version of histogram H1′. For this replication, we can use all nodes as data sources and therefore accelerate the data transfer. At t_2, we have two full distributed copies of the data sets. Assuming we monitor a long-term shift in interest in our statistics. As a consequence, we create a new histogram H2. Like at the beginning of this example, we first distribute the new histogram and start a new feeding process at time t_3. At time t_4 and time t_5, each nodes delete one of the old histograms and its data set.

8.3 Summary and Future Work

In this chapter, we described several approaches to address load balancing challenges at runtime of a community-driven data grid. We differentiated between short-term peaks and long-term trends of query distribution. For short-term load balancing, we propose a master-slave replication scheme combined with statistics monitoring. For long-term trends, we evolve our histogram and thereby adapt the histogram to shifts in the data or query distribution.

Our current master-slave replication scheme uses physical neighbors for load balancing. A further in-depth comparison between choosing physical neighbors or neighbors within the identifier space for such a replication could provide additional interesting insights. Based on the described initial replication design for community-driven data grids, more advanced replication strategies are a challenging issue. While we focused on the replication of individual regions there might be some benefits from replicating all regions of a node (“region clusters”) at once. This can reduce the management overhead and further preserve the query locality.

CHAPTER 9

Outlook and Future Challenges

Community-driven data grids provide a scalable, distributed data management solution for e-science communities. In this thesis, we described the fundamental design choices for such data grids and evaluated our infrastructure on real-life data and query workloads. In addition to simulation studies, we deployed and conducted experiments with our infrastructure in the AstroGrid-D test bed as well as in the PlanetLab framework which helped us to increase the robustness of our system. We first described the core building blocks of training, feeding, and running community-driven data grids. From that foundation we built load balancing techniques such as workload-aware training, master-slave replication, and histogram evolution to shape community-driven data grids along the changes of the communities themselves.

From our point of view, this thesis offers a starting point for collaborative researchers to actively explore and design scalable data management solutions. In the following, we give a few examples of the topics we consider interesting for future research.

In order to directly deal with the existing and envisioned scale of scientific data sets, we used a training phase to create a partitioning scheme based on the major data characteristics and predominant query patterns. This approach is viable as the majority of scientific data sets does not change once published. We envision two aspects that can be further investigated in this context: 1) Based on an initial histogram from the training phase, how can we locally adapt the partitioning and replication scheme without creating a new histogram and thus increasing the dynamics of the system, and 2) how can we achieve data and query load balancing if there is a high update rate.

Data-driven applications become an increasingly important field in academia as well as in industrial business applications. Extending the use of community-driven data grids to data mining tasks beyond pure database queries could extend the number of use cases that benefit from the scalable infrastructure. Instead of bare queries, data mining tasks would be directed towards the data sites. Challenges within that area are identifying effective communication patterns and finding good data distribution schemes.

Dealing with terabyte to petabyte-scale data sets has recently triggered interesting developments like provenance-aware database systems (Groffen et al., 2007), data-aware batch processing (Wang et al., 2009), and adaptive physical design tools (Malik et al., 2008, 2009). It would be interesting to deliver the synergies of our technique and their approach to scientific researchers. Recent work (Raicu et al., 2009, 2008) on data-aware scheduling of scientific workloads proposes interesting concepts for dynamic resource provisioning and adaptive data caching. Furthermore, systems like GrayWulf (Simmhan et al., 2009; Szalay et al., 2009) show interesting developments for designing petascale cluster environments.

Besides scalable data management, exploring the vast amounts of data visually becomes increasingly important. Both interactive visualizations using grid middleware (Polak and Kranzlmüller, 2008) as well as asynchronous collaborative visualizations (Heer et al., 2009) show interesting proposals in that area. Building virtual laboratories by integrating remote sensors and instruments (Płóciennik et al., 2008) offers new perspectives for scientific cooperations.

Our work heavily profited from the fact that we had several use cases to derive actual requirements from and that we had the opportunity to evaluate our ideas and prototypes within a real deployment. How can other computer scientists experience a similar benefit like testing their research and ideas with real data sets? From the astronomers' perspective, however, the researchers see themselves confronted with several data management solutions that hopefully ease their day-to-day data-intensive research. So how can they eventually validate these systems? Motivated by the experience that standardized benchmarks such as the TPC series[1] or XMark[2] have propelled the research in their area, we proposed a benchmark for astrophysical work benches (Nieto-Santisteban et al., 2007). This benchmark is envisioned to provide astronomers with a unified setup for testing their environments and to provide computer scientists with a specific setup to test their systems against. There are additional notable efforts such as SciDB[3] (Stonebraker et al., 2009), which focus on delivering a new database model especially aimed at scientific data management.

HiSbase, our prototype of community-driven data grids, allows e-science communities to build up decentralized and cooperative information networks and offers a framework to design histogram data structures for accommodating specific data characteristics and dominant query patterns. The histogram data structure defines a partitioning scheme to benefit from high throughput via parallelism and high cache locality and is also used as routing index for increased flexibility. Given the enormous variety of use cases and applications it is unlikely to find a single best solution. Working closely with the actual scientists is an effort that not only fosters great inter-disciplinary collaborations. Furthermore, addressing their challenges allows us to shape the data management for future e-science communities.

[1]http://www.tpc.org/
[2]http://monetdb.cwi.nl/xml/
[3]http://scidb.org/

APPENDIX A

Example Execution of the Minimum Latency Path Algorithm

In this section we will execute Algorithm 5.1 presented in Section 5.3 (page 53) on the example graph G^* depicted in Figure A.1. For the sake of simplicity, edges are labeled with integer values instead of real latency values. In this scenario, the node labeled s denotes the source. Furthermore, let v_1, v_2, v_4, and v_5 be data grid nodes or transit nodes. Assuming that node v_3 is a data grid node, we send a packet from s to v_3. In order to find a minimum latency path from s to v_3 (Figure A.2), we compute all minimum latency paths from s by executing the algorithm as shown in Figure A.3. As latency $L(v_3)$ is 5, we know that the minimum latency path from s to v_3 has a latency of 5. A depth-first search on the completely annotated graph G^* (Figure A.3(f)) yields the path that offers the lowest possible latency between s and v_3.

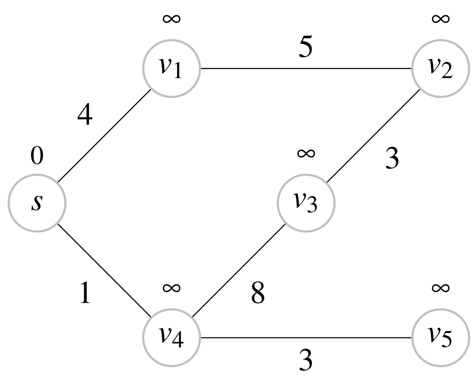


Figure A.1: Example graph G^* with latency of non-source nodes set to ∞

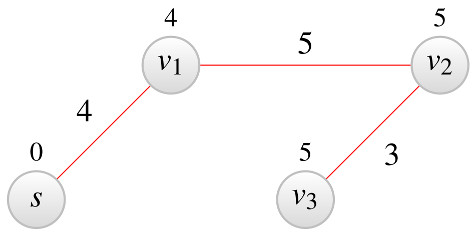
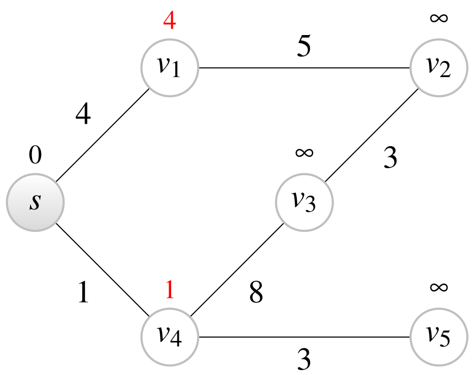
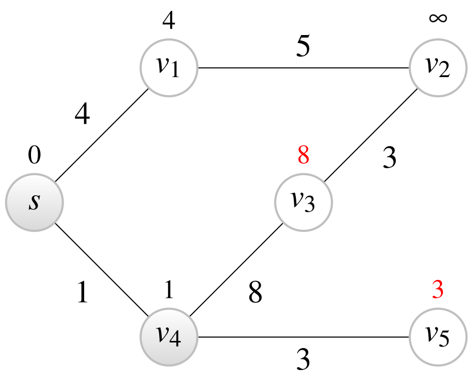


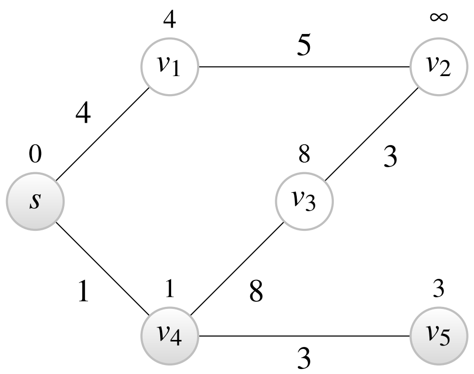
Figure A.2: Minimum latency path between node s and node v_3



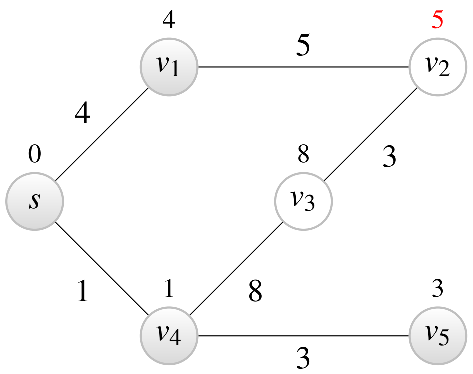
(a) Updating latency for neighbors of s



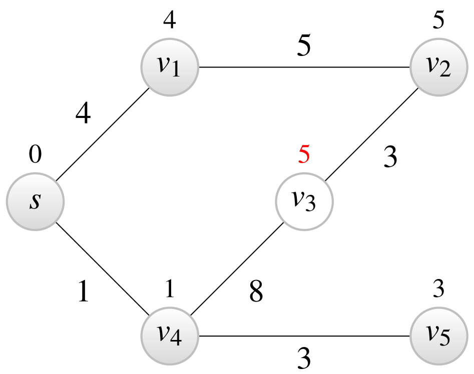
(b) Updating latency for neighbors of v_4



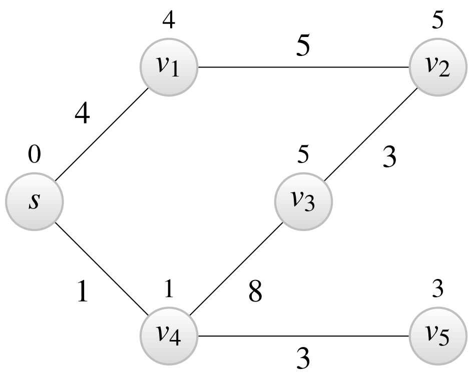
(c) Adding v_5 to S, but no labels are updated



(d) Updating latency for neighbors of v_1



(e) Updating latency for neighbors of v_2



(f) Adding v_3 to set S and the algorithm terminates as $N(G,S) \setminus S = \emptyset$

Figure A.3: Example execution of Algorithm 5.1 on G^* depicted in Figure A.1 (gray nodes are in S)

Bibliography

ABDELGUERFI, M. AND WONG, K.-F.: *Parallel Database Techniques*. Wiley-IEEE Computer Society Press, 1998, ISBN 978-0-8186-8398-5.

ABERER, K., CUDRÉ-MAUROUX, P., DATTA, A., DESPOTOVIC, Z., HAUSWIRTH, M., PUNCEVA, M., AND SCHMIDT, R.: P-Grid: a self-organizing structured P2P system. *SIGMOD Record*, 32(3):29–33, 2003.

ALLOCK, W., BRESNAHAN, J., KETTIMUTHU, R., LINK, M., DUMITRESCU, C., RAICU, I., AND FOSTER, I.: The Globus Striped GridFTP Framework and Server. In: *Proc. of the ACM/IEEE SC Conf.*, Seattle, WA, USA, November 2005.

ALY, M., MORSILLO, N., CHRYSANTHIS, P. K., AND PRUHS, K.: Zone Sharing: A Hot-Spots Decomposition Scheme for Data-Centric Storage in Sensor Networks. In: *Proc. of the Intl. Workshop on Data Management for Sensor Networks*, pp. 21–26, Trondheim, Norway, August 2005.

ALY, M., PRUHS, K., AND CHRYSANTHIS, P. K.: KDDCS: A Load-Balanced In-Network Data-Centric Storage Scheme for Sensor Networks. In: *Proc. of the ACM Intl. Conf. on Information and Knowledge Management*, pp. 317–326, Arlington, VA, USA, November 2006.

ANTONIOLETTI, M., ATKINSON, M., BAXTER, R., BORLEY, A., HONG, N. C., COLLINS, B., HARDMAN, N., HUME, A., KNOX, A., JACKSON, M., KRAUSE, A., LAWS, S., MAGOWAN, J., PATON, N., PEARSON, D., SUGDEN, T., WATSON, P., AND WESTHEAD, M.: The design and implementation of Grid database services in OGSA-DAI. *Concurrency and Computation: Practice and Experience*, 17(2-4):357–376, 2005.

ANTONIOLETTI, M., ATKINSON, M., KRAUSE, A., LAWS, S., MALAIKA, S., PATON, N. W., PEARSON, D., AND RICCARDI, G.: Web Services Data Access and Integration - The Core (WS-DAI) Specification, Version 1.0. http://www.ogf.org/documents/GFD.74.pdf, July 2006.

ASANO, T., RANJAN, D., ROOS, T., WELZL, E., AND WIDMAYER, P.: Space-filling curves and their use in the design of geometric data structures. *Theoretical Computer Science*, 181(1):3–15, July 1997.

ASPNES, J., KIRSCH, J., AND KRISHNAMURTHY, A.: Load Balancing and Locality in Range-Queriable Data Structures. In: *Proc. of ACM Symposium on Principles of Distributed Computing*, pp. 115–124, St. John's, Newfoundland, Canada, July 2004.

ASPNES, J. AND SHAH, G.: Skip Graphs. In: *Proc. of the ACM/SIAM Symposium on Descrete Algorithms*, pp. 384–393, Baltimore, MD, USA, January 2003.

AWERBUCH, B. AND LEIGHTON, T.: A Simple Local-Control Approximation Algorithm for Multicommodity Flow. In: *Proc. of the Annual Symposium on Foundations of Computer Science*, pp. 459–468, Palo Alto, CA, USA, November 1993, URL http://doi.ieeecomputersociety.org/10.1109/SFCS.1993.366841.

BANAEI-KASHANI, F. AND SHAHABI, C.: SWAM: A Family of Access Methods for Similarity-Search in Peer-to-Peer Data Networks. In: *Proc. of the ACM Intl. Conf. on Information and Knowledge Management*, pp. 304–313, Washington, DC, USA, November 2004.

BEAUMONT, O., KERMARREC, A.-M., MARCHAL, L., AND RIVIÈRE, É.: VoroNet: A scalable object network based on Voronoi tessellations. In: *Proc. of the Intl. Parallel and Distributed Processing Symposium*, pp. 1–10, Long Beach, CA, USA, March 2007.

BRAUMANDL, R., KEMPER, A., AND KOSSMANN, D.: Quality of Service in an Information Economy. *ACM Trans. on Internet Technology*, 3(4):291–333, November 2003.

BUNEMAN, P. AND TAN, W. C.: Provenance in Databases. In: *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1171–1173, Beijing, China, June 2007.

CAI, M., FRANK, M., CHEN, J., AND SZEKELY, P.: MAAN: A Multi-Attribute Addressable Network for Grid Information Services. *Journal of Grid Computing*, 2(1):3–14, March 2004.

CARLSON, A., BÖHRINGER, H., SCHOLL, T., AND VOGES, W.: Finding Galaxy Clusters using Grid Computing Technology. In: *Proc. of the IEEE Intl. Conf. on e-Science and Grid Computing (demo)*, Bangalore, India, December 2007.

CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., AND ROWSTRON, A.: Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):100–110, 2002.

COHEN, B.: Incentives Build Robustness in BitTorrent. In: *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R.: PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. of the VLDB Endowment*, 1(2):1277–1288, 2008.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C.: *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, second edition, 2001, ISBN 0-262-03293-7.

CRAINICEANU, A., LINGA, P., MACHANAVAJJHALA, A., GEHRKE, J., AND SHANMUGASUNDARAM, J.: P-Ring: An Efficient and Robust P2P Range Index Structure. In: *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pp. 223–234, Beijing, China, June 2007.

CSABAI, I., TRENCSÉNI, M., HERCZEGH, G., DOBOS, L., JÓZSA, P., PURGER, N., BUDAVÁRI, T., AND SZALAY, A.: Spatial Indexing of Large Multidimensional Databases. In: *Proc. of the Conference on Innovative Data Systems Research*, pp. 207–218, Asilomar, CA, USA, January 2007.

DABEK, F., ZHAO, B., DRUSCHEL, P., KUBIATOWICZ, J., AND STOICA, I.: Towards a Common API for Structured Peer-to-Peer Overlays. In: *International Workshop on Peer-to-Peer Systems (IPTPS), LNCS*, vol. 2, 2003.

DATTA, A., HAUSWIRTH, M., JOHN, R., SCHMIDT, R., AND ABERER, K.: Range Queries in Trie-Structured Overlays. In: *Proc. of the IEEE Intl. Conf. on Peer-to-Peer Computing*, pp. 57–66, Konstanz, Germany, August 2005.

DAVIDSON, S. B., BOULAKIA, S. C., EYAL, A., LUDÄSCHER, B., MCPHILLIPS, T. M., BOWERS, S., ANAND, M. K., AND FREIRE, J.: Provenance in Scientific Workflow Systems. *IEEE Data Engineering Bulletin*, 30(4):44–50, 2007.

DEWITT, D. J. AND GRAY, J.: Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, 1992.

DU MOUZA, C., LITWIN, W., AND RIGAUX, P.: SD-Rtree: A Scalable Distributed Rtree. In: *Proc. of the Intl. Conf. on Data Engineering*, pp. 296–305, Istanbul, Turkey, April 2007.

DU MOUZA, C., LITWIN, W., AND RIGAUX, P.: Large-scale indexing of spatial data in distributed repositories: the SD-Rtree. *VLDB Journal*, 2009, doi: 10.1007/s00778-009-0135-4.

ENKE, H., STEINMETZ, M., RADKE, T., REISER, A., RÖBLITZ, T., AND HÖGQVIST, M.: AstroGrid-D: Enhancing Astronomic Science with Grid Technology. In: *Proc. of the German e-Science Conference*, Baden-Baden, Germany, May 2007.

EVEN, S., ITAI, A., AND SHAMIR, A.: On the complexity of time table and multi-commodity flow problems. In: *Proc. of the Annual Symposium on Foundations of Computer Science*, pp. 184–193, Berkeley, CA, USA, October 1975, URL http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4567876&isnumber=4567845&punumber=4567844&k2dockey=4567876@ieeecnfs.

FINKEL, R. A. AND BENTLEY, J. L.: Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4:1–9, March 1974.

FOSTER, I. AND IAMNITCHI, A.: On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In: *International Workshop on Peer-to-Peer Systems (IPTPS), LNCS*, vol. 2, 2003.

FRANKLIN, M., HALEVY, A., AND MAIER, D.: From Databases to Dataspaces: A New Abstraction for Information Management. *SIGMOD Record*, 34(4):27–33, 2005.

GAEDE, V. AND GÜNTHER, O.: Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

GANESAN, P., BAWA, M., AND GARCIA-MOLINA, H.: Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems. In: *Proc. of the Intl. Conf. on Very Large Data Bases*, pp. 444–455, Toronto, Canada, September 2004a.

GANESAN, P., YANG, B., AND GARCIA-MOLINA, H.: One Torus to Rule them All: Multi-dimensional Queries in P2P Systems. In: *Proc. of the Intl. Workshop on the Web and Databases*, pp. 19–24, Maison de la Chimie, Paris, France, June 2004b.

GARGANTINI, I.: An Effective Way to Represent Quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.

GIETZ, P., GRIMM, C., GRÖPER, R., MAKEDANZ, S., PFEIFFENBERGER, H., SCHIFFERS, M., AND ZIEGLER, W.: A concept for attribute-based authorization on D-Grid resources. *Future Generation Computer Systems*, 25(3):275–280, March 2009, doi: 10.1016/j.future.2008.05.008.

GRAY, J., SANTISTEBAN, M. A. N., AND SZALAY, A. S.: The Zones Algorithm for Finding Points-Near-Point or Cross-Matching Spatial Datasets. Technical Report MSR-TR-2006-52, Microsoft Research, Microsoft Cooperation, Redmond, WA, USA, April 2006.

GROFFEN, F., KERSTEN, M. L., AND MANEGOLD, S.: Armada: a Reference Model for an Evolving Database System. In: *Proc. of the GI Conference on Database Systems for Business, Technology, and Web*, pp. 417–435, Aachen, Germany, March 2007.

GU, Y., GROSSMAN, R. L., SZALAY, A., AND THAKAR, A.: Distributing the Sloan Digital Sky Survey Using UDT and Sector. In: *Proc. of the IEEE Intl. Conf. on e-Science and Grid Computing*, p. 56, Amsterdam, The Netherlands, December 2006.

HAVERKORT, H. J. AND VAN WALDERVEEN, F.: Locality and Bounding-Box Quality of Two-Dimensional Space-Filling Curves. *Computing Research Repository*, abs/0806.4787, 2008.

HEER, J., VIÉGAS, F. B., AND WATTENBERG, M.: Voyagers and Voyeurs: Supporting Asynchronous Collaborative Visualization. *Communications of the ACM*, 52(1):87–97, January 2009.

HILBERT, D.: Über die stetige Abbildung einer Linie auf ein Flächenstück. *Math. Ann.*, 38:459–460, 1891.

HIRONAKA, K., SAITO, H., AND TAURA, K.: High Performance Wide-area Overlay using Deadlock-free Routing. In: *Intl. Symposium on High Performance Distributed Computing*, pp. 81–90, Munich, Germany, June 2009.

HUEBSCH, R., HELLERSTEIN, J. M., LANHAM, N., LOO, B. T., SHENKER, S., AND STOICA, I.: Querying the Internet with PIER. In: *Proc. of the Intl. Conf. on Very Large Data Bases*, pp. 321–332, Berlin, Germany, September 2003.

IVANOVA, M., NES, N., GONCALVES, R., AND KERSTEN, M.: MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database. In: *Proc. of the Intl. Conf. on Scientific and Statistical Database Management*, p. 13, Banff, Canada, July 2007.

JAGADISH, H. V., OOI, B. C., AND VU, Q. H.: BATON: a balanced tree structure for peer-to-peer networks. In: *Proc. of the Intl. Conf. on Very Large Data Bases*, pp. 661–672, Trondheim, Norway, August 2005.

KINDERMANN, S., STOCKHAUSE, M., AND RONNEBERGER, K.: Intelligent Data Networking for the Earth System Science Community. In: *Proc. of the German e-Science Conference*, Baden-Baden, Germany, May 2007.

KIRCHLER, W., SCHIFFERS, M., AND KRANZLMÜLLER, D.: Harmonizing the Management of Virtual Organizations Despite Heterogeneous Grid Middleware – Assessment of Two Different Approaches. In: *Proc. of the Cracow Grid Workshop*, pp. 245–253, Cracow, PL, October 2008.

KOSSMANN, D.: The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, December 2000.

KOTTHA, S., ABHINAV, K., MÜLLER-PFEFFERKORN, R., AND MIX, H.: Accessing Bio-Databases with OGSA-DAI – A Performance Analysis. In: *Proc. of the Intl. Workshop on Distributed, High-Performance and Grid Computing in Computational Biology*, Eilat, Israel, 2006.

KREFTING, D., BART, J., BERONOV, K., DZHIMOVA, O., FALKNER, J., HARTUNG, M., HOHEISEL, A., KNOCH, T. A., LINGNER, T., MOHAMMED, Y., PETER, K., RAHM, E., SAX, U., SOMMERFELD, D., STEINKE, T., TOLXDORFF, T., VOSSBERG, M., VIEZENS, F., AND WEISBECKER, A.: MediGRID: Towards a user friendly secured grid infrastructure. *Future Generation Computer Systems*, 25(3):326–336, March 2009, doi: 10.1016/j.future.2008.05.005.

KROMPASS, S., AULBACH, S., AND KEMPER, A.: Data Staging for OLAP- and OLTP-Applications on RFID Data. In: *Proc. of the GI Conference on Database Systems for Business, Technology, and Web*, pp. 542–561, Aachen, Germany, March 2007.

KUNTSCHKE, R., SCHOLL, T., HUBER, S., KEMPER, A., REISER, A., ADORF, H.-M., LEMSON, G., AND VOGES, W.: Grid-based Data Stream Processing in e-Science. In: *Proc. of the IEEE Intl. Conf. on e-Science and Grid Computing*, p. 30, Amsterdam, The Netherlands, December 2006.

LEDLIE, J., SHNEIDMAN, J., SELTZER, M., AND HUTH, J.: Scooped, Again. In: *International Workshop on Peer-to-Peer Systems (IPTPS), LNCS*, vol. 2, 2003.

MALIK, T. AND BURNS, R.: Workload-Aware Histograms for Remote Applications. In: *Data Warehousing and Knowledge Discovery*, pp. 402–412, Turin, Italy, September 2008.

MALIK, T., BURNS, R., AND CHAUDHARY, A.: Bypass Caching: Making Scientific Databases Good Network Citizens. In: *Proc. of the Intl. Conf. on Data Engineering*, pp. 94–105, Tokyo, Japan, April 2005.

MALIK, T., BURNS, R., CHAWLA, N. V., AND SZALAY, A.: Estimating Query Result Sizes for Proxy Caching in Scientific Ddatabase Federations. In: *Proc. of the ACM/IEEE SC Conf.*, p. 36, Tampa, FL, USA, November 2006.

MALIK, T., WANG, X., BURNS, R., DASH, D., AND AILAMAKI, A.: Automated Physical Design in Database Caches. In: *Proc. of the Intl. Conf. on Data Engineering Workshops*, pp. 27–34, Cancun, Mexico, April 2008.

MALIK, T., WANG, X., DASH, D., CHAUDHARY, A., AILAMAKI, A., AND BURNS, R.: Adaptive Physical Design for Curated Archives. In: *Proc. of the Intl. Conf. on Scientific and Statistical Database Management*, pp. 148–166, New Orleans, LA, USA, June 2009.

NAUMANN, F., BILKE, A., BLEIHOLDER, J., AND WEIS, M.: Data Fusion in Three Steps: Resolving Schema, Tuple, and Value Inconsistencies. *IEEE Data Engineering Bulletin*, 29(2):21–31, 2006.

NIETO-SANTISTEBAN, M. A., GRAY, J., SZALAY, A. S., ANNIS, J., THAKAR, A. R., AND O'MULLANE, W. J.: When Database Systems Meet the Grid. In: *Proc. of the Conference on Innovative Data Systems Research*, pp. 154–161, Asilomar, CA, USA, January 2005.

NIETO-SANTISTEBAN, M. A., SCHOLL, T., KEMPER, A., AND SZALAY, A.: 20 Spatial Queries for an Astronomer's Bench(mark). In: *Proc. of the Astronomical Data Analysis Software & Systems Conference*, London, UK, September 2007.

O'MULLANE, W., LI, N., NIETO-SANTISTEBAN, M., SZALAY, A., THAKAR, A., AND GRAY, J.: Batch is back: CasJobs, serving multi-TB data on the Web. In: *Proc. of the Intl. Conf. on Web Services*, pp. 33–40, Orlando, FL, USA, July 2005.

ORENSTEIN, J. AND MERRETT, T.: A Class of Data Structures for Associative Searching. In: *Proc. of the ACM SIGACT-SIGMOD Symp. on Principles of Database Sys.*, pp. 181–190, Waterloo, Ontario, Canada, April 1984.

ÖZSU, M. T. AND VALDURIEZ, P.: *Principles of Distributed Database Systems*. Prentice-Hall, 1999.

PAPADOMANOLAKIS, S. AND AILAMAKI, A.: AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In: *Proc. of the Intl. Conf. on Scientific and Statistical Database Management*, pp. 383–392, Stantorini Island, Greece, June 2004.

PENTARIS, F. AND IOANNIDIS, Y.: Query Optimization in distributed Networks of Autonomous Database Systems. *ACM Trans. on Database Systems*, 31(2):537–583, June 2006.

PITOURA, T., NTARMOS, N., AND TRIANTAFILLOU, P.: Replication, Load Balancing, and Efficient Range Query Processing in DHT Data Networks. In: *Proc. of the Intl. Conf. on Extending Database Technology*, pp. 131–148, Munich, Germany, March 2006.

PLANTIKOW, S., PETER, K., HÖGQVIST, M., GRIMME, C., AND PAPASPYROU, A.: Generalizing the data management of three community grids. *Future Generation Computer Systems*, 25(3):281–289, March 2009, doi: 10.1016/j.future.2008.05.001.

PŁÓCIENNIK, M., ADAMI, D., BARCELÓ, Á. D. G., COZ, I. C., DAVOLI, F., GAMBA, P., KELLER, R., KRANZLMÜLLER, D., LABOTIS, I., MEYER, N., MONTEOLIVA, A., PRICA, M., PUGLIESE, R., SALON, S., SCHIFFERS, M., WATZL, J., ZAFEIROPOULOS, A., AND DE LUCAS, J. M.: DORII – Deployment of Remote Instrumentation Infrastructure. In: *Proc. of the Cracow Grid Workshop*, pp. 78–85, Cracow, PL, October 2008.

POLAK, M. AND KRANZLMÜLLER, D.: Interactive videostreaming visualization on grids. *Future Generation Computer Systems*, 24(1):39–45, January 2008, doi: 10.1016/j.future.2007.03.006.

POOSALA, V., IOANNIDIS, Y. E., HAAS, P. J., AND SHEKITA, E. J.: Improved Histograms for Selectivity Estimation of Range Predicates. In: *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pp. 294–305, Montreal, Quebec, Canada, June 1996.

RAHM, E. AND BERNSTEIN, P. A.: A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

RAICU, I., FOSTER, I., SZALAY, A., AND TURCU, G.: AstroPortal: A Science Gateway for Large-scale Astronomy Data Analysis. In: *Proc. of the TeraGrid Conf.*, June 2006.

RAICU, I., FOSTER, I. T., ZHAO, Y., LITTLE, P., MORETTI, C. M., CHAUDHARY, A., AND THAIN, D.: The Quest for Scalable Support of Data-Intensive Workloads in Distributed Systems. In: *Intl. Symposium on High Performance Distributed Computing*, pp. 207–216, Munich, Germany, June 2009.

RAICU, I., ZHAO, Y., FOSTER, I., AND SZALAY, A.: Accelerating Large-Scale Data Exploration through Data Diffusion. In: *Proc. of the Intl. Workshop on Data-Aware Distributed Computing*, pp. 9–18, Boston, MA, USA, June 2008.

RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S.: A Scalable Content-Addressable Network. In: *Proc. of the ACM SIGCOMM Intl. Conf. on Data Communication*, pp. 161–172, 2001.

REHN, J., BARRASS, T., BONACORSI, D., HERNANDEZ, J., SEMENIOUK, I., TUURA, L., AND WU, Y.: PhEDEx high-throughput data transfer management system. In: *Proc. of the Intl. Conf. on Computing in High Energy and Nuclear Physics*, Mumbai, India, February 2006.

ROWSTRON, A. I. T. AND DRUSCHEL, P.: Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In: *Proc. of the IFIP/ACM Intl. Conf. on Distributed Systems Platforms (Middleware)*, pp. 329–350, Heidelberg, Germany, November 2001.

SALLES, M. A. V., DITTRICH, J.-P., KARAKASHIAN, S. K., GIRARD, O. R., AND BLUNSCHI, L.: iTrails: Pay-as-you-go Information Integration in Dataspaces. In: *Proc. of the Intl. Conf. on Very Large Data Bases*, pp. 663–674, Vienna, Austria, September 2007.

SAMET, H.: *The Design and Analysis of Spatial Data Structures*. Addison Wesley, 1990, ISBN 0-201-50255-0.

SAMET, H.: Hierarchical Representations of Collections of Small Rectangles. *ACM Computing Surveys*, 20(4):271–309, December 1998.

SAMET, H.: *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006, ISBN 0-12-369446-9.

SCHOLL, T., BAUER, B., GUFLER, B., KUNTSCHKE, R., REISER, A., AND KEMPER, A.: Scalable community-driven data sharing in e-science grids. *Future Generation Computer Systems*, 25(3):290–300, March 2009a, doi: 10.1016/j.future.2008.05.006.

SCHOLL, T., BAUER, B., GUFLER, B., KUNTSCHKE, R., WEBER, D., REISER, A., AND KEMPER, A.: HiSbase: Histogram-based P2P Main Memory Data Management. In: *Proc. of the Intl. Conf. on Very Large Data Bases (demo)*, pp. 1394–1397, Vienna, Austria, September 2007a.

SCHOLL, T., BAUER, B., KUNTSCHKE, R., WEBER, D., REISER, A., AND KEMPER, A.: HiSbase: Informationsfusion in P2P Netzwerken. In: *Proc. of the GI Conference on Database Systems for Business, Technology, and Web (demo)*, pp. 602–605, Aachen, Germany, March 2007b.

SCHOLL, T., BAUER, B., MÜLLER, J., GUFLER, B., REISER, A., AND KEMPER, A.: Workload-Aware Data Partitioning in Community-Driven Data Grids. In: *Proc. of the Intl. Conf. on Extending Database Technology*, pp. 36–47, Saint-Petersburg, Russia, March 2009b.

SCHOLL, T., GUFLER, B., MÜLLER, J., REISER, A., AND KEMPER, A.: P2P-Datenmanagement für e-Science-Grids. *Datenbank-Spektrum*, 8(26):26–33, September 2008.

SCHOLL, T. AND KEMPER, A.: Community-Driven Data Grids. *Proc. of the VLDB Endowment*, 1(2):1672–1677, 2008.

SCHOLL, T., KUNTSCHKE, R., REISER, A., AND KEMPER, A.: Community Training: Partitioning Schemes in Good Shape for Federated Data Grids. In: *Proc. of the IEEE Intl. Conf. on e-Science and Grid Computing*, pp. 195–203, Bangalore, India, December 2007c.

SCHOLL, T., REISER, A., AND KEMPER, A.: Collaborative Query Coordination in Community-Driven Data Grids. In: *Intl. Symposium on High Performance Distributed Computing*, pp. 197–206, Munich, Germany, June 2009c.

SCHÜCKER, P., BÖHRINGER, H., AND VOGES, W.: Detection of X-ray Clusters of Galaxies by Matching RASS Photons and SDSS Galaxies within GAVO. *Astronomy & Astrophysics*, 420:61–74, 2004.

SHU, Y., OOI, B. C., TAN, K.-L., AND ZHOU, A.: Supporting Multi-dimensional Range Queries in Peer-to-Peer Systems. In: *Proc. of the IEEE Intl. Conf. on Peer-to-Peer Computing*, pp. 173–180, Konstanz, Germany, August 2005.

SILBERSTEIN, A., COOPER, B. F., SRIVASTAVA, U., VEE, E., YERNENI, R., AND RAMAKRISHNAN, R.: Efficient Bulk Insertion into a Distributed Ordered Table. In: *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pp. 765–778, Vancouver, Canada, June 2008.

SIMMHAN, Y., BARGA, R., VAN INGEN, C., NIETO-SANTISTEBAN, M., DOBOS, L., LI, N., SHIPWAY, M., SZALAY, A. S., WERNER, S., AND HEASLEY, J.: GrayWulf: Scalable Software Architecture for Data Intensive Computing. In: *Hawaii Intl. Conference on System Sciences*, Waikoloa, HI, USA, January 2009.

SINGH, V., GRAY, J., THAKAR, A., SZALAY, A., RADDICK, J., BOROSKI, B., LEBEDEVA, S., AND YANNY, B.: SkyServer Traffic Report – The First Five Years. Technical Report MS-TR-2006-190, Microsoft Research, Microsoft Cooperation, Redmond, WA, USA, December 2006.

SPRINGEL, V., WHITE, S. D. M., JENKINS, A., FRENK, C. S., YOSHIDA, N., GAO, L., NAVARRO, J., THACKER, R., CROTON, D., HELLY, J., PEACOCK, J. A., COLE, S., THOMAS, P., COUCHMAN, H., EVRARD, A., COLBERG, J., AND PEARCE, F.: Simulating the joint evolution of quasars, galaxies and their large-scale distribution. *Nature*, 435:629–636, June 2005.

STOICA, I., MORRIS, R., KARGER, D. R., KAASHOEK, M. F., AND BALAKRISHNAN, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In: *Proc. of the ACM SIGCOMM Intl. Conf. on Data Communication*, pp. 149–160, San Diego, CA, USA, August 2001.

STONEBRAKER, M., BECLA, J., DEWITT, D., LIM, K.-T., MAIER, D., RATZESBERGER, O., AND ZDONIK, S.: Requirements for Science Data Bases and SciDB. In: *Proc. of the Conference on Innovative Data Systems Research*, Asilomar, CA, USA, January 2009.

SZALAY, A. S., BELL, G., VANDENBERG, J., WONDERS, A., BURNS, R., FAY, D., HEASLEY, J., HEY, T., NIETO-SANTISTEBAN, M., THAKAR, A., VAN INGEN, C., AND WILTON, R.: GrayWulf: Scalable Clustered Architecture for Data Intensive Computing. In: *Hawaii Intl. Conference on System Sciences*, Waikoloa, HI, USA, January 2009.

TANIN, E., HARWOOD, A., AND SAMET, H.: Using a distributed quadtree index in peer-to-peer networks. *VLDB Journal*, 16:165–178, February 2007.

VENUGOPAL, S., BUYYA, R., AND RAMAMOHANARAO, K.: A Taxonomy of Data Grids for Distributed Data Sharing, Management, and Processing. *ACM Computing Surveys*, 38(1):3, March 2006.

WANG, X., BURNS, R., AND MALIK, T.: LifeRaft: Data-Driven, Batch Processing for the Exploration of Scientific Databases. In: *Proc. of the Conference on Innovative Data Systems Research*, Asilomar, CA, USA, January 2009.

WANG, X., BURNS, R., AND TERZIS, A.: Throughput-Optimized, Global-Scale Join-Processing in Scientific Federations. In: *Intl. Workshop on Networking Meets Databases*, Cambridge, UK, April 2007.

WITTEN, I. AND FRANK, E.: *Data Mining*. Morgan Kauffmann, second edition, 2005, ISBN 0-12-088407-0.

YILDIRIM, E., YIN, D., AND KOSAR, T.: Balancing TCP Buffer vs Parallel Streams in Application Level Throughput Optimization. In: *Proc. of the Intl. Workshop on Data-Aware Distributed Computing*, pp. 21–30, Munich, Germany, June 2008.

ZHAO, B., HUANG, L., STRIBLING, J., RHEA, S., JOSEPH, A., AND KUBIATOWICZ, J.: Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

ZIMMERMANN, R., KU, W.-S., WANG, H., ZAND, A., AND BARDET, J.-P.: A Distributed Geotechnical Information Management and Exchange Architecture. *IEEE Internet Computing*, 10(5):26–33, 2006.