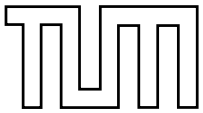

TECHNISCHE
UNIVERSITÄT
MÜNCHEN

Distributed Query Processing for Locality-Aware Data in P2P Networks

Daniel Weber

Diplomarbeit in Informatik

Technische Universität München
Fakultät für Informatik

Diplomarbeit in Informatik

Distributed Query Processing for Locality-Aware Data in P2P Networks

Daniel Weber

Aufgabensteller: Prof. Alfons Kemper, Ph.D.
Betreuer: Dipl.-Inf. Tobias Scholl
Abgabedatum: 20. Februar 2007

Eidesstattliche Erklärung

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Garching, am 20. Februar 2007 .....

Daniel Weber

Abstract

Over the years, the growth of storage capacities, computing performance, and the distribution of the Internet have lead to new possibilities in sciences like astrophysics. Large telescope arrays and sensor networks can record petabytes of data. Due to the exponential growth, this data cannot be processed at once and on site. So-called e-sciences share their distributed resources, cooperate in processing their data, and gain new results by cross-correlating data from different sources. As an alternative to centralized approaches, we have developed HiSbase, a prototype supporting astrophysical analysis and cross-correlation of multiple spatial catalogues. Based on a peer-to-peer system, numerous computers distributed all over the world can participate in HiSbase sharing not only storage capacity or computing resources, but also their main memory. Scientists can query the spatial catalogues at any participating node without having to regard the distribution of the data within the HiSbase network. In addition, a spatial cross-match allows them to easily correlate information from different sources.

Contents

List of Figures

Listings

Chapter 1

Motivation

The beginnings of the Internet have been shaped by universities and research organizations sharing their computing resources. Over the years, the growth of storage capacities, computing performance, and the distribution of the Internet have lead to new possibilities in some sciences like astrophysics, geophysics, climatology, or meteorology. Telescopes and sensors can be combined to form large telescope arrays and sensor networks which can record petabytes of data. Due to the exponential growth, this data can no longer be processed at once and on site. This lead to the development of so-called *e-sciences* where research groups share their distributed resources, cooperate in processing their data, and gain new results by cross-correlating data obtained from different sources. Centralized approaches like data warehouses used for processing and cross-correlation suffer from expensive data shipping to the centralized site, huge data volumes not even partially fitting into main memory, and an overwhelming amount of queries. As an alternative, decentralized grid-based systems can offer better performance. A grid coordinates resources at decentralized locations and under decentralized control using standardised and open interfaces to provide nontrivial qualities of services [Fos02]. In contrast to the centralized approach where data is collected at a central site for computing and the results are distributed afterwards, the grid approach distributes the computing resources of all participants among them.

Using grid technologies, we want to develop HiSbase, a prototype supporting astrophysicists in analyzing and cross-correlating multiple catalogues with spatial data. These catalogues are recorded by various telescopes and satellites measuring different spectra and use the equatorial coordinate system for object positioning: the right ascension (RA) in horizontal direction from 0° up to 360° and the declination (DEC) in vertical direction from −90° up to 90° as shown in figure 1.1. Based on a peer-to-peer (P2P) system[1], numerous computers distributed all over the world will be able to participate in HiSbase sharing not only storage capacity or computing resources, but also their main memory. In addition, we will support cross-correlation by implementing a spatial cross-match which provides scientists with a near-a-point search in several catalogues. This allows researchers to easily combine information from different sources with a certain error of measurement and to gain new results from an improved point of view.

1.1 Typical data distribution in e-sciences: Data skew

A typical phenomenon in natural sciences like astrophysics is that the observed objects are not uniformly distributed. On the one hand there are clusters of densely populated

[1] Several P2P implementations will be discussed in chapter 2.

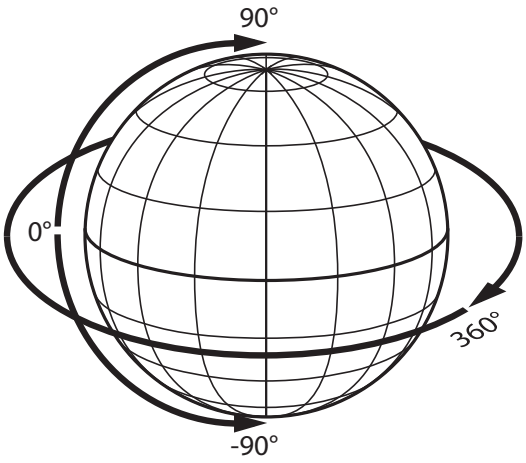

Figure 1.1: The equatorial coordinate system

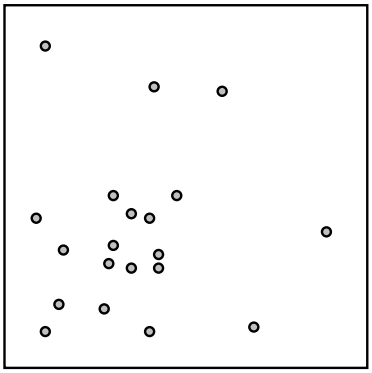

Figure 1.2: Astrophysical data with irregular distribution

regions; on the other hand there are wide and nearly unpopulated[2] regions. An example of such skewed data is shown in figure 1.2. This so-called *data skew* will be handled by HiSbase in a way that allows us to evenly distribute the data over the participating nodes by the use of *equi-depth histograms*. These histograms will have to be created from a representative data set[3] before setting up the network. In contrast to the traditional usage of peer-to-peer systems as distributed hash tables (DHT), we will utilize the histogram as a hash function to map the spatial objects on logical regions.

1.2 Typical query patterns: Region-based queries

Region-based queries are another typical feature supported by HiSbase. This means that queries include a predicate which focuses them at a specified region which is of special interest for the scientist. Examples of such scenarios are meteorology where weather conditions like rainfall or atmospheric pressure of a country or continent could be of

[2]At least scientists currently do not know about objects within these regions.
[3]The representative data set can be a randomly chosen subset of the already mentioned catalogues.

interest, astrophysics where the radiation output and wavelengths of certain spatial objects or regions are analyzed, or geology where the regional occurrence and concentration of minerals might be in the focus of researchers.

1.3 Increasing throughput

The first major goal of the HiSbase project is to realize an increased throughput within a HiSbase network consisting of several nodes compared to a single HiSbase node. This is done by distributing the data on all participating nodes while preserving locality. In traditional peer-to-peer implementations each data object would be individually mapped onto the peer-to-peer system which in our case destroys locality. HiSbase maps spatial regions onto the peer-to-peer system and hence preserves locality. The distribution of the data among all nodes reduces the amount of data managed by a single node and therefore increases the chance that repeatedly queried data can be held in main memory. In addition, the same spatial regions from different catalogues are mapped onto the same node and therefore HiSbase can cross-correlate different catalogues directly on the nodes holding the data.

Chapter 2

Overview on relevant technologies

This chapter will give you a short overview on the major components and technologies we considered or used for HiSbase. We will start with a discussion of some peer-to-peer networks which will be used as the infrastructure for communication in HiSbase. Then we will have a glance at the topic of space-filling curves which represent data structures necessary for the distribution of data within the project. After that we will address a method for spatial cross-matching within SQL which is an essential part for the applicability in the astrophysics community. Finally we will focus on some database management systems considered as database backend for the project.

2.1 Peer-to-peer networks

Since about 1999, several peer-to-peer networks have been developed, mostly as a platform for file-sharing via the Internet. The peer-to-peer model was a new approach to networking, since it did not conform to the established client-server models like a web browser retrieving web pages from a server. In a peer-to-peer network, each participating node is a server as well as a client.

The existing peer-to-peer networks can roughly be divided into a first generation of peer-to-peer-systems like Kazaa [KAZ07], Gnutella [GDF07], and Freenet [Cla99] and a second generation like CAN [RFH+01], Chord [SMK+01], Pastry [RD01], and Tapestry [ZHS+04]. Some publications use a three-generation model with Napster being a representative of the first generation, but Napster uses a hierarchical approach with an index-server used by all participating nodes and therefore does not fit into a plain peer-to-peer schema.

While the first generation of peer-to-peer systems has widely spread due to several file-sharing networks, the second generation focuses on efficient[1] data structures and algorithms. The evolution of peer-to-peer systems leads to the possibility of sharing the distributed computing resources among all participants, which is of great interest for the e-science community. A major portion of the success of peer-to-peer systems is also due to their self-organizing architecture, allowing them to compensate outages and congestion of peers or links.

Every peer-to-peer system is based on two essential protocols:

- network management protocol
- data management protocol

[1] Efficiency in the sense of reducing bandwidth utilization and the number of hops to reach a certain node.

The network management protocol allows nodes to join and leave the peer-to-peer network. For joining a network, the new node has to know at least one node already participating in the network. This node will be used as a so-called *bootstrap node* and will receive a join request from the new node. The join request initiates the reorganisation of the neighbourhood and the distributed index. In most peer-to-peer systems, there is no method for gracefully[2] leaving the net, because outages usually occur without prior announcement. Hence the network management protocol is also responsible for detection of nodes leaving the network and again the reorganisation of the neighbourhood and the index.

The data management protocol allows peers to send, forward, and receive messages and — using such messages — to add, remove, and look up data in the peer-to-peer network.

Peer-to-peer systems basically appear in three different flavours [DZD+03]:

- as distributed hash table (DHT) supporting addition, removal, and lookup of data referenced by keys
- as distributed object location and routing (DOLR) mechanism allowing message delivery to a node nearby and not exactly matching an object's key
- as group multicast or anycast providing scalable communication and coordination in large and highly dynamic groups

In the following sections, we will focus on the second generation of peer-to-peer systems due to their improved efficiency and we will limit the discussion to Chord, Pastry, and Tapestry, because they have a comparable key space. CAN on the other hand uses an n-dimensional torus as key space and therefore has not been considered for the HiSbase project.

2.1.1 Chord

Chord assigns keys, respectively IDs — calculated with the SHA-1 algorithm [EJ101] to ensure consistent hashing — to each participating node and to each stored object. These keys are mapped onto a one-dimensional circular key space with at maximum 2^m entries. For simple routing, each node knows its direct successor, i.e. the next node clockwise on the ring. This information is already sufficient to support routing to any participating node. A message to a key travels from the sender over the successor to the successor's successor and so on until it reaches the destination. The destination is either the node having the exact key or the node having the smallest key which is still greater than the message's destination key.

To reduce the number of hops during the travel of a message, the nodes store additional routing information, the so-called *finger table*, with up to a limited amount of entries. Each entry i contains a pointer to the node with distance $2^{(i-1)}$; the distance between two entries in the finger table grows at the power of 2. Therefore the number of hops a message has to visit in a network consisting of n nodes before reaching the destination is $O(\log\ n)$ with high probability.

When a message is sent to a specific key, Chord searches the finger table for the node with the highest key preceding the destination key and routes the message to that node, which — if necessary — repeats the routing step or processes the message itself.

[2] To gracefully leave a net means to follow a specified protocol for terminating the connection to the other nodes.

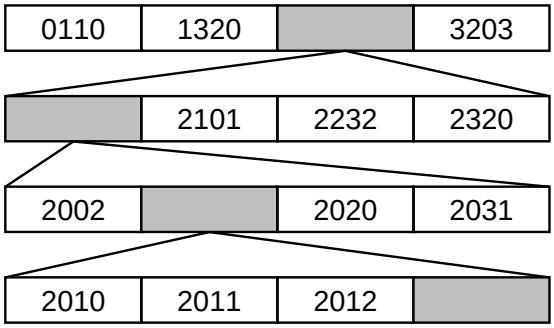

Figure 2.1: Simplified Pastry routing table

2.1.2 Pastry

Pastry again uses a one-dimensional circular key space as Chord does. Messages can be routed to any key in the key space. The destination of a message is either the node with the specified key or the node with the key closest to the destination key.

In contrast to Chord, Pastry maintains three tables with routing information and a routing metric which represents the "distance" between nodes, e.g. the round trip time of messages.

- The *leaf set* contains l entries, with $l/2$ being the neighbouring nodes clockwise on the ring and the other $l/2$ being the neighbouring nodes counter clockwise on the ring.
- The *neighbourhood set* represents the m closest nodes in terms of the routing metric. It is used to maintain locality in the routing process.
- The *routing table* consists of $\lceil log_{2^b} N \rceil$ rows with $2^b - 1$ entries in each row. All entries at row n refer to nodes whose IDs share the first n digits with the local node's ID but whose $(n+1)^{th}$ digit has one of the $2^b - 1$ possible values other than the $(n+1)^{th}$ digit in the local node's ID. A simplified illustration of a Pastry routing table for a node with ID 2013 is shown in figure 2.1.

When a message is sent to a specified key, the node first searches the leaf set for a matching node and directly delivers the message to that node. If the search in the leaf set was not successful, the node searches its routing table for another node whose key shares a longer prefix with the destination key than the node itself does. If this also fails, a node from the routing table whose key has the same prefix length but is numerically closer is selected.

2.1.3 Tapestry

Tapestry is an approach closely related to Pastry, having some differences. Like Pastry, Tapestry uses a prefix routing mechanism but without considering an additional metric to improve routing between certain nodes.

When a message is sent to a specified key, the sending node considers the leftmost digit and chooses another node having the same leftmost digit as the next hop. Intermediate nodes first determine the i leading digits they have in common with the destination key and then choose a node with $i + 1$ common digits as the next hop.

As a special improvement to distributed object location and routing systems (DOLR), Tapestry uses the publish mechanism to store a copy of the object to be published at each node on the way between the publishing node and the node maintaining the assigned key. These copies increase the speed of later object look-ups, when the look-up message is routed over one of the nodes having a local copy. This mechanism is called *surrogate routing*.

2.1.4 The Common API

Each of the mentioned peer-to-peer systems has a different API and they have been developed with slightly different requirements in mind. As a result, applications being developed based on one of the peer-to-peer systems cannot be easily migrated to another system unless the developer provides an abstraction layer. The Common API [DZD+03] is an approach to define such an abstraction layer and allow applications based on the Common API to change the underlying peer-to-peer system without serious changes to the code. Due to the different orientations of the various peer-to-peer systems — some focus on their application as DHT, others as DOLR — the Common API attaches not on top of those systems but on the so-called *key-based routing (KBR)* layer, which represents the common denominator of all peer-to-peer systems.

2.2 Space-filling curves

A detailed discussion of space-filling curves would be far beyond the scope of this work. In this section, we will focus on some special space-filling curves and on their applicability to the HiSbase project. For extensive information on space-filling curves we recommend [Sag94].

At the end of the 19th century, Giuseppe Peano (1858-1932) first described space-filling curves as continuous curves completely filling an n-dimensional space. Usually, each space-filling curve consists of a generator curve whose sections can be repeatedly replaced by scaled copies of itself which leads to a more and more detailed space-filling curve. Examples of such generator curves in 2-dimensional space and the resulting space-filling curves after some steps of iteration are shown in figure 2.2, 2.3, and 2.4. As we currently need space-filling curves in HiSbase only for the linearisation of a 2-dimensional space, the following examples will be limited to non-overlapping continuous curves on planar surfaces.

2.2.1 Hilbert curve

David Hilbert (1862-1943) was the first to describe a general geometric generating algorithm [Hil91] which covers a whole class of space-filling curves.

"If the interval I can be mapped continuously onto the square Q, then after partitioning I into four congruent subintervals and Q into four congruent subsquares, each subinterval can be mapped continuously onto one of the subsquares. Next, each subinterval is, in turn, partitioned into four congruent subintervals and each subsquare into four congruent subsquares, and the argument is repeated." [Sag94]

With the space-filling curve shown in figure 2.2, Hilbert demonstrated that the subsquares can be arranged in such a way that adjacent intervals correspond to adjacent

subsquares and that all subintervals of an interval I correspond to subsquares of a square corresponding to I.

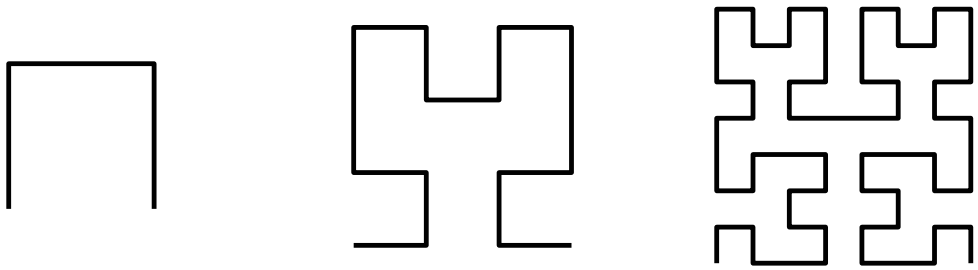

Figure 2.2: Hilbert curve with three iteration steps

Another space-filling curve based on Hilbert's curve has been described by Eliakim Hastings Moore (1862-1932). Moore's version of the Hilbert curve is shown in figure 2.3 and demonstrates the possibility of ending the curve at the same point where it started while Hilbert's curve ended at the opposite of the starting edge.

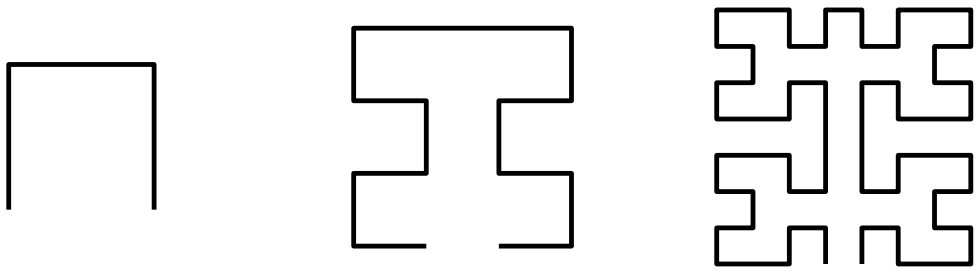

Figure 2.3: Moore's version of the Hilbert curve with three iteration steps

It is easy to see that both versions require changing the orientation of the generator curve during the iterations.

2.2.2 Z-curve

Another class of space-filling curves has been described by Henri Leon Lebesgue (1875-1941). The Lebesgue- or Z-curve does not require subsquares corresponding to adjacent intervals to be adjacent themselves. This can easily be seen in figure 2.4 where the right-upper square of the generator curve shares no edge with the left-lower square but both squares correspond to adjacent intervals.

Note that the Z-curve in contrast to Hilbert's and Moore's curves does not require changing the orientation of the generator curve during the iterations.

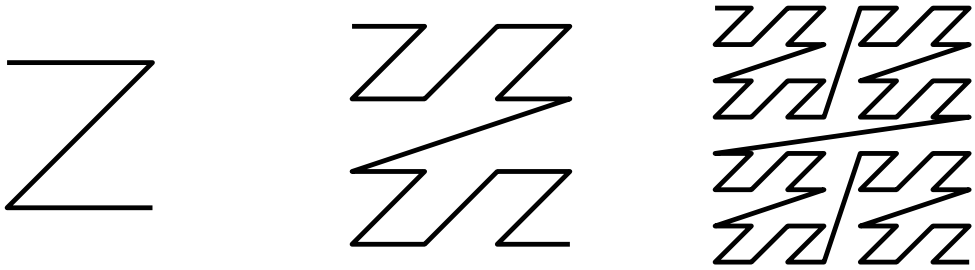

Figure 2.4: Z-curve with three iteration steps

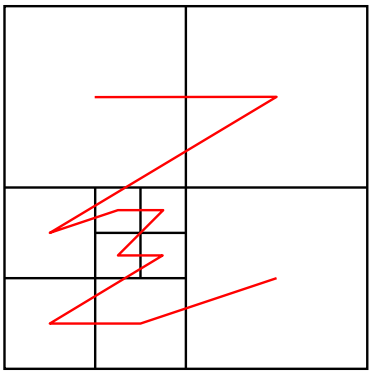

Figure 2.5: Z-quadtree

2.2.3 Z-quadtree

In 1974, Raphael A. Finkel and Jon Louis Bentley introduced a data structure called *quadtree* [FB74] to support composite database indexes which are of special interest for applications processing spatial data. Based on the idea of using a data structure like search trees for two-dimensional data, quadtrees recursively partition the data space into four disjoint spaces. The recursion for each partition will continue until a certain condition, e. g. when the number of data objects within the partition falls below a specified limit, is met. The resulting regions of the quadtree can then be linearised by applying the Z-curve ordering algorithm as shown in figure 2.5. A detailed description of the implementation and usage of quadtrees in the HiSbase project is available in [Bau07].

2.3 Spatial cross-matches

As already mentioned in chapter 1, the ability to correlate data from various catalogues is of special interest for astrophysicists. This allows scientists e. g. to consider several radiation spectra when categorizing a newly found spatial object and hence determine a bunch of physical attributes of the object at once instead of having to gain this information step by step. Due to variances between all catalogues resulting e. g. from atmospheric

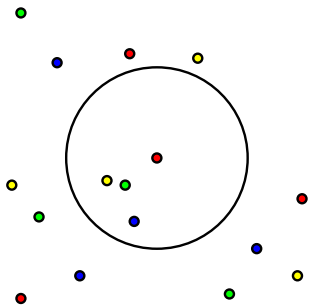

Figure 2.6: Near-a-point search around the object in the centre

flickering, different weather conditions, or imprecise mechanics, it is not possible to use an exact match algorithm for cross-matching. Instead, one has to do a near-a-point search to find objects within a certain distance to the initial object in other catalogues. Intuitively, this near-a-point search could be done by selecting all objects within a specified radius around the initial object as shown in figure 2.6, but this does not work using equatorial coordinates because the RA-axis gets condensed the closer the DEC-axis gets to 90° or −90°, respectively. Instead, we can use Cartesian coordinates on the unit sphere to avoid this problem.

Due to the processing effort of cross-matching outside of SQL, Jim Gray et. al. [GNSS06] introduced a method for cross-matching objects without leaving SQL. In addition, it is expensive to do complex mathematic operations on a bigger data set. Hence their algorithm works in three steps as shown in figure 2.7. It tries to narrow down the amount of data within the first two steps by limiting the equatorial RA- and DEC-ranges to a rectangle around the initial point with basically an edge having twice the length of the cross-match radius. See lines 4 and 5 of listing 2.1 for the limitation of the RA- and DEC-ranges. In the last step, the cross-match switches to Cartesian coordinates on the unit sphere and fetches all objects left within the cross-match radius with the expression in line 6 of listing 2.1.

```
1 select ...
2    from catalogue1 as c1 left outer join catalogue2 as c2
3    on
4       abs(c1.ra-c2.ra) < r and
5       abs(c1.dec-c2.dec) < r and
6       (c1.x*c2.x+c1.y*c2.y+c1.z*c2.z) < cos(radians(r))
```

Listing 2.1: A simple cross-matching algorithm without leaving SQL

Please note that for a “real world cross-match” the RA-range has to be stretched when the DEC-range gets closer to the polar regions. This leads to the following SQL expression describing a cross-match with a modified first step at line 4 to 7:

```
1 select ...
2    from catalogue1 as c1 left outer join catalogue2 as c2
3    on
4       abs(c1.ra-c2.ra) < case when abs(c1.dec)+r > 89.9 then 180
5                            else degrees(abs(atan(sin(radians(r)) /
```

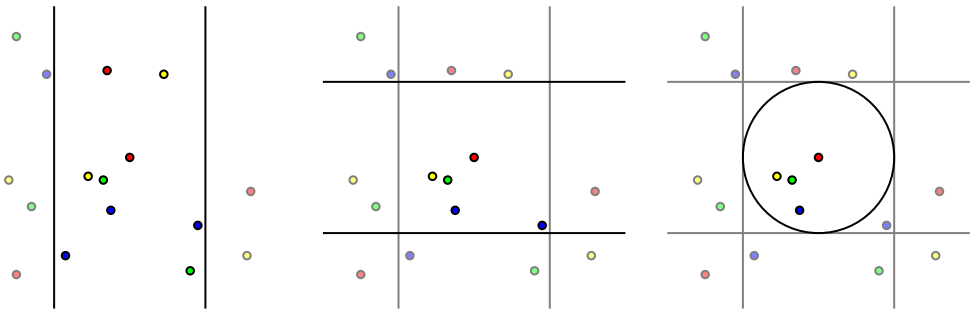

Figure 2.7: Cross-match algorithm in three steps

```
6                                   sqrt(abs(cos(radians(c1.dec-r)) *
7                                   cos(radians(c1.dec+r)))))) end and
8       abs(c1.dec-c2.dec) < r and
9       (c1.x*c2.x+c1.y*c2.y+c1.z*c2.z) < cos(radians(r))
```

Listing 2.2: A cross-matching algorithm with correct handling of polar regions

2.4 Database management systems

Although the database management system used within HiSbase is one of its main components, we will not provide an in-depth discussion, because HiSbase is meant to be independent of specific database management systems and SQL implementations. The necessity of a database management system results from the demand for a standardised relational query language which is provided by SQL and the ability to store large data sets and allow efficient access to them.

The following database management systems have been taken into consideration during the HiSbase project:

- **HSQLDB**
  HSQLDB [HSQ07] is a relational database management system based on Java [SJ07]. It can operate as a standalone database server as well as a database integrated into other Java-based applications. Tables can be created as temporary tables being held only in memory or as persistent tables with different methods of storage and caching. The major drawback of HSQLDB 1.7.3 is the unavailability of "less than" or "greater than" operators in JOIN-clauses, which makes it impossible to implement the spatial cross-match discussed in section 2.3.
- **IBM DB2**
  IBM DB2 [DB207] is a commercial relational database management system. Several versions are offered for different commercial deployment scenarios. In 2006, IBM added the free version "IBM DB2 Express-C" which is available at `http://www-306.ibm.com/software/data/db2/express/`. IBM DB2 operates as a standalone database server and cannot be integrated into Java applications.
- **Apache Derby**
  Apache Derby [ADD07] is another relational database management system implemented in Java. Like HSQLDB, it can be run standalone or be embedded in other

Java-based software. A restriction on the complexity of WHERE-clauses has shown up as a disadvantage in the HiSbase project, as it limits the number of regions a single HiSbase node could handle. We did not yet manage to determine the exact number of conditions in the WHERE-clause which is allowed with Apache Derby.

Due to the drawbacks and disadvantages of HSQLDB and our experiences with IBM DB2 during previous projects, we decided to use IBM DB2 during the first steps of development. Nevertheless it is one goal of the HiSbase project to allow different database management systems. A setup with several HiSbase nodes using Apache Derby is considered for the near future.

Chapter 3

Architecture and design of HiSbase

The following chapter will present details on the architecture and the design of HiSbase, especially the steps necessary for data staging and query processing.

3.1 Preparations

Before diving into the preparation of the HiSbase system, we have to define some assumptions concerning the data we want to handle with the HiSbase project. HiSbase is developed as an application supporting astrophysicists, so we will support astrophysical catalogues. The data in each catalogue has to include RA- and DEC-values according to the equatorial coordinate system to support the query windows which have already been mentioned in chapter 1. Furthermore, the catalogues also have to include X, Y and Z values for each object which represent the Cartesian coordinates on the unit sphere. These values are necessary for the cross-matching algorithm. If these values are not included in a catalogue, they can be calculated, using the following SQL command [Sch05]:

```
1 update catalogue set
2    x = cos(radians(dec)) * cos(radians(ra)),
3    y = cos(radians(dec)) * sin(radians(ra)),
4    z = sin(radians(dec))
```

Listing 3.1: Calculating Cartesian coordinates from equatorial coordinates

Before being used in HiSbase, the catalogues have to be made available in a database archive supporting SQL. In our case, we will utilize IBM DB2 [DB207] as the database management system on the archive as well as on each of the HiSbase nodes. After importing and preparing the spatial catalogues, we are ready to define the distribution of the data on our HiSbase nodes.

3.1.1 Creating the histograms

As already mentioned in chapter 1, HiSbase is a distributed application based on a P2P system. The P2P system of our choice has been FreePastry [FP07] because of the possibility of using a proximity metric which can improve routing performance.

Traditional P2P applications use a distributed hash table (DHT) for mapping objects on keys, IDs, or nodes. The application of a traditional hash function on objects specified by equatorial coordinates would possibly lead to objects with neighbouring coordinates being distributed among several nodes. In our case, this is not useful because we want to preserve the spatial locality of our astrophysical data. This locality allows us to reduce

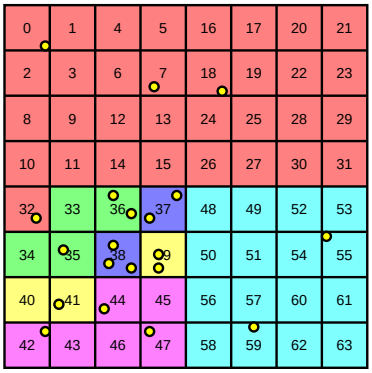

Figure 3.1: Region generation with Z-curve intervals

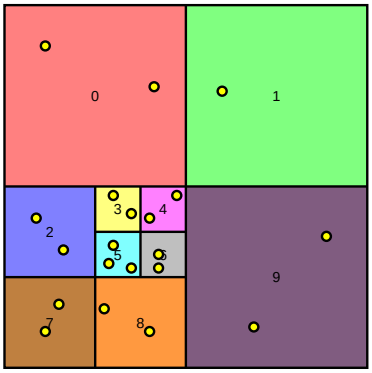

Figure 3.2: Region generation with a Z-quadtree

the number of HiSbase nodes involved in a query in contrast to a traditional DHT where a query would require to be processed on several nodes. As replacement for the DHT we use a histogram structure which partitions the space into regions. The regions are either generated using Z-curve intervals as shown in figure 3.1 or using the leaves of a Z-quadtree as shown in figure 3.2.

To handle the data skew already mentioned in chapter 1, the chosen histogram technique has to be trained using a representative data set. During this training phase, we want to realize an even distribution of the data among all regions. Ideally, we would get a histogram having several regions with each of them comprising approximately the same amount of data. After distributing this histogram to each HiSbase node, we can launch the HiSbase network by starting a first node and using it as bootstrap node for the others.

Note that the regions as well as the HiSbase nodes are mapped onto the same P2P key space (see figure 3.3) and therefore both are assigned IDs. FreePastry's routing mechanism allows us to send messages to a region instead of the node maintaining the region. This is necessary because in the distributed HiSbase environment a node neither knows what regions are assigned to the other nodes nor which other nodes beyond its neighbourhood exist.

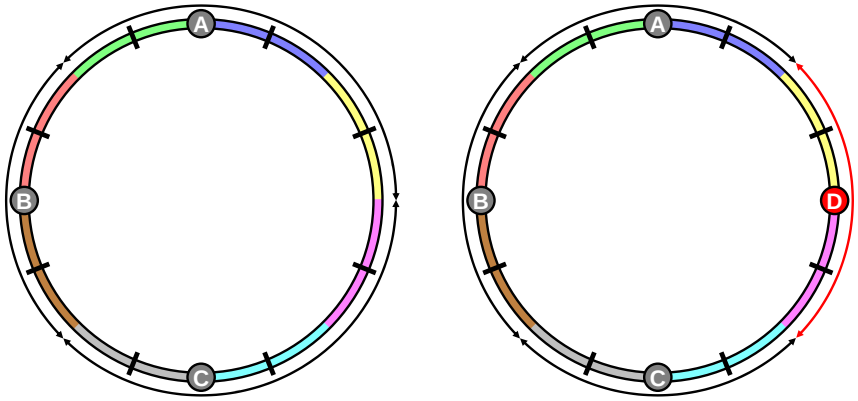

Figure 3.3: A pastry ring with 8 regions, initially 3 nodes and one newly joining node

3.1.2 Staging the data

Now that we have a HiSbase network, we need to equip the nodes with the data. The data staging is not only necessary after start-up but also when the assignment of regions to nodes changes at runtime. This is caused by other nodes joining or leaving the HiSbase network.

To stay up-to-date concerning the assignment of regions, each of the nodes monitors its leaf set which is automatically updated by the P2P system during joins or leaves. If a node experiences a change within its leaf set, it has to discover if there has been a change to the set of regions assigned to the node, as shown in figure 3.3. In that case, the node has to recalculate the borders of its regions, connect to the database archive, and retrieve the data lying within the calculated borders from the subscribed catalogues. To prevent this possibly time-consuming task from occurring too often, the nodes will wait some time after each change within the leaf set, because e. g. one node leaving the system might rejoin within seconds or might be replaced by another node. Only if – after the timeout – no additional changes have taken place, the further steps in the staging process will be executed. During data staging, the HiSbase network is able to accept and process queries, but the current implementation cannot ensure complete results.

3.2 Query processing

3.2.1 Installing and preparing the query

Once all participating HiSbase nodes have been equipped with their histograms, have joined the system, and have retrieved the data belonging to their regions, HiSbase is ready for processing queries at each of the nodes. A newly issued query is first parsed at the local node to determine the query's RA- and DEC-range. To realise this in an efficient manner, we require RA and DEC to be specified within the WHERE clause using the BETWEEN keyword. This allows us to easily extract the RA- and DEC-range.

```
1 select ...
2   from ...
3   where ra between r1 and r2
```

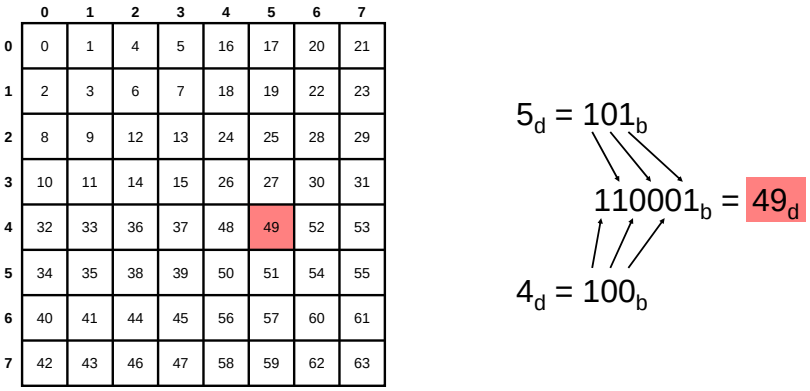

Figure 3.4: Calculating Z-indices with bit interleaving

```
4     and dec between d1 and d2
```

Listing 3.2: Allowed SQL-queries in HiSbase

To support wrap-around queries[1] we have to enhance the syntax. When the given query has $r_2 < r_1$, we can assume a wrap-around query. Due to the restrictions defined in the SQL-specifications, we have to resolve the query after the detection of a wrap-around query:

```
1 select ...
2   from ...
3   where ( ra between r1 and 360 or ra between 0 and r2 )
4     and dec between d1 and d2
```

Listing 3.3: Resolved wrap-around SQL-queries

Given the RA- and DEC-range, the node calculates all histogram regions affected by the query. A histogram region is affected by the query if the intersection between the histogram regions translated into a spatial area and the RA- and DEC-range of the query is not empty. The way how the affected histogram regions are calculated depends on the histogram type.

When using Z-curve histograms, we have to scale down the spatial area to the histogram's resolution and can compute the Z-index with bit interleaving (see figure 3.4) done on the scaled coordinates. Using these Z-indices, we can test the boundaries of each histogram region and detect whether the individual histogram region is affected by the query or not.

When using Z-quadtree histograms, the calculation of the affected histogram regions is a top-down approach. In each step we test the area represented by the current Z-quadtree histogram node for an intersection with the queried area defined by the RA- and DEC-range. If there is an intersection and the current Z-quadtree histogram node is also a leaf, the leaf is added. If it is no leaf, the whole test will be repeated for all quadrants of the node. As result, we will get all Z-quadtree histogram leaves having intersections with the queried area.

[1] Wrap-around queries leave the spatial coordinate system at one end and re-enter it at another end, e. g. with $ra = [359°, 1°]$.

Now that we know the affected histogram regions, we choose one of the regions to be the coordinator for the query processing. At this level of HiSbase we do not talk about nodes, because we do not know which node is responsible for a specified region. The node, being responsible for the region chosen as coordinator, will of course do the coordination task, not the region.

3.2.2 Coordinating the query

After the coordinator election has taken place, the coordinator will receive a query message from the node that has been accepting the query initially. This query message also contains the set of affected regions. The coordinator now initializes its buffers necessary to keep the individual parts of the result sets until all data has been retrieved. Then the query message will be forwarded to each affected region, regardless of being handled by the same or different nodes. During query processing — the steps necessary for query processing are described in the next section — the coordinator waits for answer messages from the affected nodes and stores them in its buffers until all outstanding answers have been received. Finally an answer message containing all the buffered results from the affected nodes is sent back to the node that has been accepting the query initially.

3.2.3 Processing the query

When the query message reaches an affected region, the node maintaining the region can hand off the query contained in the message to the local SQL database management system. The SQL database management system processes the query and returns a result set. This result set is then sent back to the coordinator as the payload of an answer message.

If the originally issued query affects more than one region, it is likely that one node receives the same query message more than once because each node may maintain more than one region. To handle this case, we keep track of already processed queries to prevent unnecessary processing. This is done by saving a timestamp and a hash-value of the query expression, before it is handed off to the database system. If timestamp and hash-value are already known on the local node, the query message is identified as a duplicate and consequently ignored. In addition, the coordinator has to be informed about the *weight* of an answer message. The weight is the number of regions which are represented in the answer message. It is important for the coordinator, because he might send out multiple query messages to the same node but will only receive one answer message. The weight allows the coordinator to detect when all affected regions have sent their parts of the answer.

Chapter 4

Implementation of the prototype

After we have laid out the architecture and the design of the HiSbase project in chapter 3, we will focus on the implementation in this chapter. We will start with the P2P layer of HiSbase which will provide the infrastructure for the communication between the HiSbase nodes. This will be followed by a discussion of the HiSbase message hierarchy being used for query processing and, after that, by a description of query processing itself. Finally we will focus on the implementation of the data staging mechanism.

4.1 The P2P infrastructure

As already mentioned in chapter 3, we decided to utilize FreePastry [FP07] developed at Rice University (Houston, Texas) as the P2P substrate for the HiSbase project. The major portion of the P2P infrastructure will be provided by FreePastry. We will have to supply at least a so-called *driver* class (the class `HiSbase` in figure 4.1) which is in java terminology our application and a so-called *application* class (the class `HiSbaseApp` in figure 4.1) which handles the messaging within HiSbase. In addition we need a special node ID factory (the class `StaticNodeIdFactory` in figure 4.1) to allow our nodes to regain recently assigned IDs.

4.1.1 The driver class

The driver class implements the main method of the HiSbase application. It is the main method's duty to configure the FreePastry environment — in our case especially the time

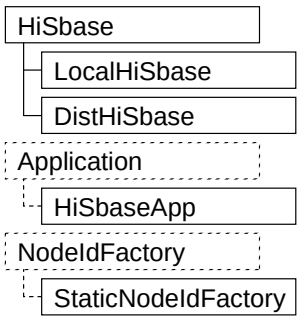

Figure 4.1: The P2P-infrastructure in HiSbase

source, the logging and the file containing the FreePastry parameters —, instantiate a new driver object using the configured environment and let the driver object create at least one new pastry node object.

During the development of HiSbase we have implemented two different driver classes:

- `DistHiSbase`
  A driver class for a P2P system with single nodes on different computers in a distributed environment communicating via TCP/IP.
- `LocalHiSbase`
  A driver class for a P2P system with multiple nodes within a single java virtual machine on one computer.

Both driver classes have been derived from the class `HiSbase` which supplies the common attributes and methods used in the subclasses.

4.1.2 The application class

The class `HiSbaseApp` implements the `Application` interface provided by FreePastry. At least three methods have to be implemented in this class to provide the basic messaging functionality:

- `deliver`
  The `deliver` method is invoked when an incoming message is intended for the local node.
- `forward`
  The `forward` method is called when an incoming message has to be routed by the local node to the destination node.
- `update`
  The `update` method is invoked when another node has either joined or left the neighbourhood set.

In our case, the `deliver` method verifies that the received message is of the correct type and does the further processing according to the visitor pattern. It calls the message's `process` method which has to be available in all messages derived from the `HiSbaseMsg` class. A detailed discussion of the different kinds of messages will follow in section 4.2. The `forward` method just returns `true` to allow all of our messages to be routed over all nodes. The `update` method is currently only used for diagnostic purposes to keep track of other nodes joining and leaving the neighbourhood set.

4.1.3 The node ID factories

For typical projects, FreePastry either assigns the node IDs at random or based on the IP address of the node. For HiSbase, the assignment of IDs is of special interest, because it determines how many histogram regions will be maintained by a node. To distribute the histogram regions nearly equally among all nodes, it is useful to assign IDs at regular intervals to the nodes. Neither the random node ID factory nor the IP address-based node ID factory support these regular intervals.

In addition, the random node ID factory also does not ensure the reassignment of the same ID to a node when the node has to restart. Assigning a new ID to an "old" node

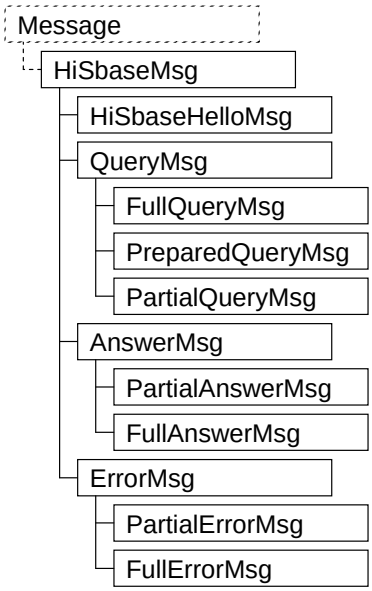

Figure 4.2: The message hierarchy of HiSbase

forces a change to the set of assigned regions on up to five nodes: The node rejoining the HiSbase network, the two former neighbours, and the two new neighbours. All these five nodes would have to go through the data staging process which consumes time.

This leads to the implementation of a static node ID factory which allows us to configure a permanent node ID that will survive reboots of a node and therefore minimize the number of nodes affected by changes to the region assignment.

4.2 Message exchange

As already mentioned in section 4.1.2, the `HiSbaseApp` class is responsible for message routing and delivery. To simplify the implementation of the message processing at the destination nodes, we derived our own abstract base class `HiSbaseMsg` from FreePastry's `Message` class. The `HiSbaseMsg` class defines the abstract method `process`, which allows the `HiSbaseApp`'s `deliver` method to check the incoming message object for being an instance of the `HiSbaseMsg` class and – if so – just call the `process` method as mentioned in section 4.1.2. In addition, the `HiSbaseMsg` adds an attribute and `get/set` methods to retrieve the node handle of the node originally sending the message.

Based on the `HiSbaseMsg` class, a set of message classes as shown in figure 4.2 has been derived. The `HiSbaseHelloMsg` has only been implemented for diagnostic purposes during development. The `QueryMsg`, `AnswerMsg`, and `ErrorMsg` are abstract base classes for the transport of queries, answers to queries and errors during query processing. Due to the steps in message processing which are pictured in figure 4.3 and 4.4 on page 27, each of them has to be implemented in at least two flavours: The `Full...Msg` message above processing by the coordinating node and the `Partial...Msg` message beneath processing by the coordinating node. In addition, the query message also has to be implemented in a

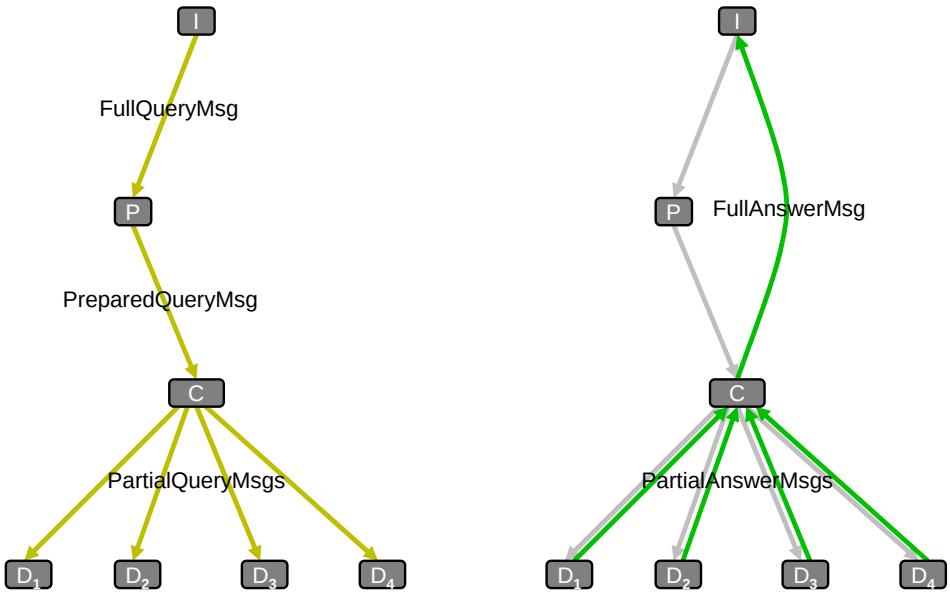

Figure 4.3: Steps during successful processing of a query. I is the node where the query has been installed, at node P (which is usually identical to I) the query has been prepared, C is the coordinator (and usually identical to D_1) and D_i are the nodes involved in processing the query.

third flavour as `PreparedQueryMsg` occurring after the preparation of the query but before coordination.

4.3 Handling queries

In HiSbase queries can be installed at each participating node. We will call this process *injection* and in figure 4.3 the node where a query is injected is denominated I. I now constructs a `FullQueryMsg` object containing the query, a query ID consisting of the query's hash value and I's node handle as source.

4.3.1 Query preparation

The `FullQueryMsg` constructed at I is then processed by the `prepareQuery` method of the `Coordinator` class[1]. Currently this preparation step also takes place at I, but this may be changed in the project's future, so in figure 4.3 we call the node in the preparation step P. The `prepareQuery` method extracts the RA/DEC window from the query and translates it into a set of regions covered by the query. Using the set of covered regions, we then choose the coordinator C which will query all covered regions individually and wait for their results to merge them and return the complete result. Please note that we always route `QueryMsg` objects to regions and not to nodes, because on the one hand FreePastry does not know the IDs of all nodes and on the other hand a HiSbase node does not know the assignment of regions to the other nodes. A HiSbase node does not even know the total number of nodes participating in the HiSbase network.

[1]Note that the `Coordinator` class not only holds the query coordination task but also all other steps of the message exchange during query processing.

In addition, during preparation, we have to rewrite the cross-match predicate `XMATCH(catalogue1,catalogue2,error)` within the issued query expression as shown in listing 4.1 into native SQL. This allows us to directly execute the query expression as explained in section 4.3.3.

```
query = query.replaceAll(
   "\\s*xmatch\\(\\s*([a-zA-Z]+[0-9]*)\\s*,\\s*([a-zA-Z]+[0-9]*)" +
   "\\s*,\\s*([0-9]+.?[0-9]*)\\s*\\)\\s*",
   "␣(abs($1.ra-$2.ra)<case␣when␣abs($1.dec)+$3␣>␣89.9␣then␣180" +
   "␣else␣degrees(abs(atan(sin(radians($3))␣/" +
   "␣sqrt(abs(cos(radians($1.dec-$3))␣*"+
   "␣cos(radians($1.dec+$3)))))))␣end␣and␣abs($1.dec-$2.dec)<$3" +
   "␣and␣($1.x*$2.x+$1.y*$2.y+$1.z*$2.z)<cos(radians($3)))␣"
);
```

Listing 4.1: Rewriting the cross-match predicate with a regular expression in Java

```
query = query.replaceAll(
   "ra\\s+between\\s+(\\+?[0-9]+(?:\\.[0-9]+)?)\\s+" +
   "and\\s+(\\+?[0-9]+(?:\\.[0-9]+)?)",
   "(␣ra␣between␣0␣and␣$2␣or␣ra␣between␣$1␣and␣360␣)␣"
);
```

Listing 4.2: Rewriting wrap-around queries with a regular expression in Java

For the coordinator election, we first determine the intersection of the set of regions covered by the query and the set of regions maintained at P. If the resulting set is not empty, we know that P maintains one of the queried regions and hence P should do the coordination of the further processing itself (now denoted as C in figure 4.3) to reduce communication between HiSbase nodes. Otherwise, we elect the node maintaining the first region covered by the query as coordinator C. A more sophisticated coordinator election will be discussed in chapter 7.

After electing the coordinator C, a `PreparedQueryMsg` object is constructed containing the same attributes as the `FullQueryMsg` object before and additionally the set of covered regions. The `PreparedQueryMsg` is then sent to C.

4.3.2 Query coordination

Arriving at C, the `PreparedQueryMsg` is processed by the `coordinateQuery` method of the `Coordinator` class. The coordinator C is liable for querying all regions covered by the query, collecting their results, and finally merging the results and sending them back to I. In order to be able to collect the results of all regions, C has to remember the number of regions covered by a query. This is done in a `HashMap` called `queryRegions` which maps the query's ID to the number of regions with still due results.

Initially the cardinality of the set of regions covered by the query is assigned to the query's ID in the map. C then constructs a `PartialQueryMsg` with the attributes of the `PreparedQueryMsg`. The former message's source attribute becomes a new attribute `originalSource` and the node handle of C will become the new source attribute. The resulting `PartialQueryMsg` is then sent to each of the regions covered by the query, which usually results in one query being sent multiple times to one node as each node usually maintains several regions. In the example in figure 4.3 the nodes receiving the

`PartialQueryMsg` are called D_1 to D_4, in the further text we will use D_i, because HiSbase is not limited in the number of nodes participating in a query.

4.3.3 Query processing

Assuming that the node D_i has received a `PartialQueryMsg`, this will be processed by the `processQuery` method of the `Coordinator` class. As already mentioned in section 4.3.2 we have to expect the same `PartialQueryMsg` to be received multiple times. Hence, we will use a HashMap called `seenQueries` which maps the IDs of the already received queries to the timestamp of their first reception. This allows us to detect repetitions of the same query and to silently ignore the duplicates. To keep the HashMap within an acceptable size, it is cleaned up during each `processQuery` call. During this clean-up, all entries belonging to queries received before a configurable amount of time (and therefore expected to be completely processed and delivered back to C) will be removed from the HashMap. In the unlikely case that an entry has been removed too early, we might receive duplicate results.

Now, after reducing the chance of processing a query more than once, we create a new `QueryProcessor` thread for the received `PartialQueryMsg` and start it. This thread at first calculates the number of local regions covered by the query – the so-called *weight* – which is necessary later on as an attribute of the `PartialAnswerMsg`. Then it calls the `queryDatabase` method of the `DbManager` class, which returns a `SerializableResultSet` object. The `SerializableResultSet` class allows us to serialize the data of a `java.sql.ResultSet` object. Using the `SerializableResultSet` object, the query ID and the already mentioned weight, a `PartialAnswerMsg` is then constructed and sent to the source of the `PartialQueryMsg` which is again C.

4.3.4 Collecting answers

The `PartialAnswerMsg` from D_i to C will be processed by the `coordinateAnswer` method of the `Coordinator` class. At first, the query ID of the answer message is compared to all query IDs known by `queryRegions`. As already mentioned in section 4.3.2, `queryRegions` maps the IDs of the queries which are currently being processed to the number of regions with outstanding results. If the query ID of the answer message is not found in `queryRegions` we can silently discard the `PartialAnswerMsg` because the corresponding query has already been completed.

Assuming that the `PartialAnswerMsg` has not been discarded, we need to determine if it has been the first answer or one of the subsequent answers. In the latter case the `SerializableResultSet`s of the previous answers have been buffered in the `partialAnswerBuffer` HashMap. The buffered results can be retrieved by using the query ID as key to the HashMap. After that, we will append the received answer to the buffered results using the `appendResultSet` method of the `SerializableResultSet` class.

Now we need to determine if we have already received all outstanding answers. This is done by retrieving the number of still outstanding regions from the `queryRegions` HashMap which has been introduced in section 4.3.2 and decrementing its value by the value of the weight attribute (mentioned in section 4.3.3) of the received `PartialAnswerMsg`. If after decrementing the value reaches zero we have received all `PartialAnswerMsg`s for our query and are now able to create a `FullAnswerMsg` object equipped with all answers

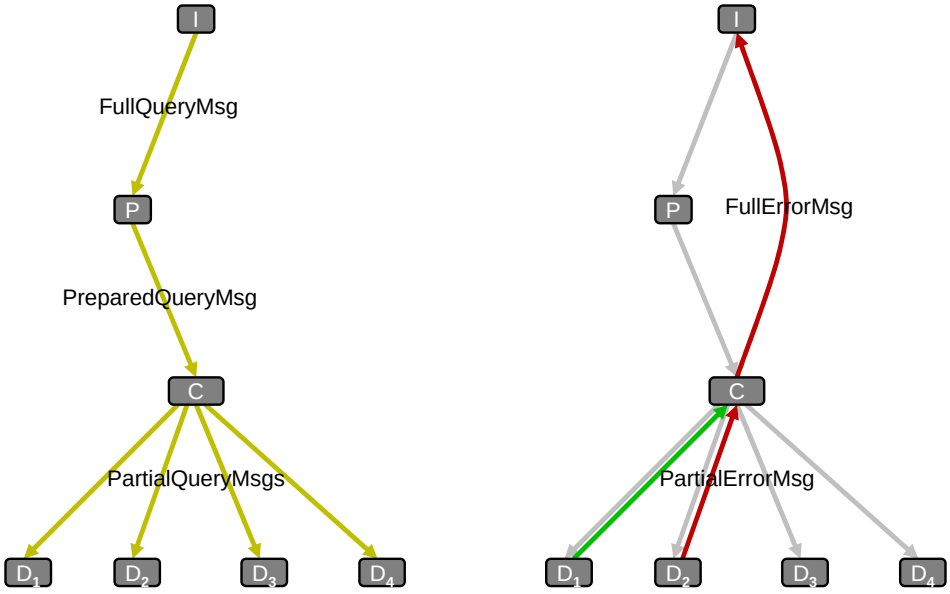

Figure 4.4: Steps during erroneous processing of a query

from the buffered `SerializableResultSet` and send the message back to the node I where the query has been injected at the beginning. In addition, we can clean up both HashMaps `queryRegions` and `partialAnswerBuffer`.

After receiving the `FullAnswerMsg` at node I, the current implementation just writes the `SerializableResultSet` to the standard output. More sophisticated options for the presentation or formatting of results will be discussed in chapter 7.

4.3.5 Handling errors

Despite of having a functioning setup for the processing of queries in HiSbase, we also have to implement some error handling during SQL-query processing. Without error handling, a user injecting a faulty query into HiSbase would have to wait infinitely for his answer. Furthermore the memory consumption in the various HashMaps of the `Coordinator` class would increase with each error because errors on certain nodes prevent us from receiving all expected answers and hence from sending the `FullAnswerMsg` and cleaning up the HashMaps.

Errors can occur on the nodes D_i during SQL-query processing in case of syntactic problems, malfunctions of the database management system installed at node D_i, or problems with the JDBC driver accessing the database. In these cases, the `QueryProcessor` thread on D_i will catch a `SQLException` or `ClassNotFoundException`, write an error message to the node's log file and send a `PartialErrorMsg` including the query ID and the error message back to the coordinating node C.

Arriving at C the `PartialErrorMsg` is processed by the `coordinateError` method of the `Coordinator` class. At first it checks if the query causing the error is one of the queries currently being coordinated. This is done by searching the query ID as key in the `queryRegions` HashMap. If there is no matching entry, the query is unknown to C which indicates that it has already been aborted by another `PartialErrorMsg`. In this case we can silently discard the `PartialErrorMsg`.

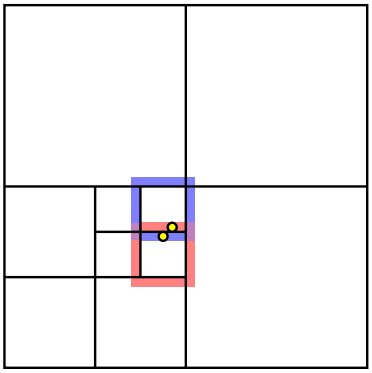

Figure 4.5: Two spatial objects lying next to a border between regions

If there is a matching entry which means that the query is (still) known to C, we cancel further coordination of the query by removing the entries in `queryRegions` and `partialAnswerBuffer` having the query's ID as key. After that, a `FullErrorMsg` again containing the query ID and the error message is sent to the node I which has injected the query.

4.4 Data staging

The process of data staging has already been mentioned in chapter 3 with a focus on the detection of changes to the set of regions maintained by the local node and hence the necessity to run through the data staging process.

In this section we will focus on the steps after the decision whether to re-stage or not. The real work is done by the `StagingManager` class and its `updateLocalData` method. The `updateLocalData` method loops over the set of regions maintained by the node. For each of the regions, it creates a `SpatialArea`[2] object and calls its `getSQLConditionString` method which returns a string with a SQL condition describing the region in equatorial coordinates.

To support cross-correlation, we had to implement an additional functionality within the previous step. As already mentioned in section 4.3.3, the query processing and hence the cross-correlation is done on each node itself. Due to the variances between different catalogues, we can run into situations where objects from different catalogues which should get caught by cross-correlation will be placed in different regions and on different nodes as shown in figure 4.5. This would lead to missing results depending on the queried areas. To handle this problem, the `getSQLConditionString` adds a frame of configurable size to each region and therefore the data objects next to borders will also be transferred to the local node during the staging process. This allows each node to do the cross-correlation independently from the other nodes. In the worst case this approach will lead to duplicate results for the same objects, which can easily be filtered during further processing.

The individual SQL conditions returned from `getSQLConditionString` are then combined to one large SQL condition describing all the node's regions. An example for a SQL

[2]The SpatialArea class is a geometry helper class providing translation operations from HiSbase regions to Cartesian coordinates and vice versa.

condition representing the regions 5, 6, and 7 in figure 4.6 is shown in listing 4.3.

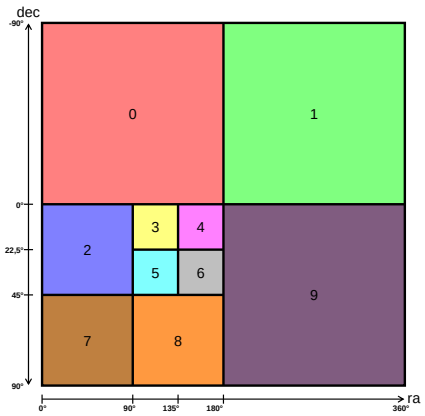

Figure 4.6: The regions of a HiSbase system before translation to SQL conditions

```
1 select *
2    from catalogue
3    where ra between 90 and 135 and dec between 22.5 and 45
4           and ra between 135 and 180 and dec between 22.5 and 45
5           and ra between 0 and 90 and dec between 45 and 90
```

Listing 4.3: SQL condition representing a set of regions

After that, another loop over all spatial catalogues calls `updateLocalCatalog` from the `DbManager` class with the catalogue's name and the SQL condition string as parameters. The `updateLocalCatalog` method connects to the database archive and queries it for the catalogue with the spatial areas described by the SQL condition string and then calls `populateCatalog` with the catalogue's name and the result set of the query just mentioned. The `populateCatalog` method connects to the local database, empties the table representing the currently processed catalogue, and then adds the data from the result set in configurable chunks. We only add the data in chunks because this allows us to prevent congesting the logs of the local database.

Please note that the data staging process can be time consuming depending on the size of the catalogues and the region assignments of the participating nodes. During the staging, a HiSbase node currently is already able to accept and process queries but the results will not be complete.

Chapter 5

Evaluation

The major goal of the HiSbase project is to realize an increased throughput of queries. In this chapter we will first describe the evaluation setup used for measuring the throughput. Then we will present and discuss the results of the evaluation.

5.1 Setup

For the evaluation, we first had to decide which scenarios should be compared and which amount of data should be handled in each scenario depending on the available equipment.

We decided to compare a HiSbase network with 16 nodes running on consumer class PCs each with 0.5 GB of memory to a standalone HiSbase node running on a server class computer with 2 GB of memory. This allowed us to utilize the server class computer as database archive for the 16 nodes during the staging process and hence reuse the same data. Each computer has been equipped with IBM DB2 Version 8.2.1.

Subsets of the SDSS [SDS07] (84,323,326 entries), ROSAT [VAB+99] (25,617,873 entries), and 2MASS [SCS+06] (28,445,694 entries) catalogues have been chosen as data, imported into separate tables on the database archive, and staged onto the 16 nodes. The tables for these catalogues bear a cluster index on the columns (RA, DEC) to support faster queries with conditions on these attributes. In addition we used the DB2 command *runstats* to create full statistics on these tables which allows the DB2 optimizer to reduce the access times again.

The necessary histogram has been created using a data set gained with the DB2 *tablesample* command from all three catalogues as shown in listing 5.1. Due to our good experiences with the Z-quadtree histograms, we preferred them over the Z-curve. The resulting Z-quadtree histogram contains 208 regions.

```
1 select ra, dec from sdss tablesample bernoulli (0.1)
2 union
3 select ra, dec from twomass tablesample bernoulli (0.1)
4 union
5 select ra, dec from rosat tablesample bernoulli (0.1)
```

Listing 5.1: Extraction of tablesamples from subsets of SDSS, ROSAT, and 2MASS

During the first start-up of the network containing 16 nodes, we used the automatic ID assignment. Later on we reassigned the nodes to the previous IDs, in order to not repeat the time-consuming staging process which has not been subject of our evaluation.

The set of queries for our evaluation has again been gained with the tablesample functionality of DB2. We extracted randomly chosen objects from the ROSAT and 2MASS

Node	first region	last region	regions
1	0	13	14
2	14	28	15
3	29	43	15
4	44	53	10
5	54	69	16
6	70	85	16
7	86	95	10
8	96	106	11
9	107	119	13
10	120	133	14
11	134	146	13
12	147	154	8
13	155	169	15
14	170	184	15
15	185	197	13
16	198	207	10

Table 5.1: Regions assigned to the 16 nodes

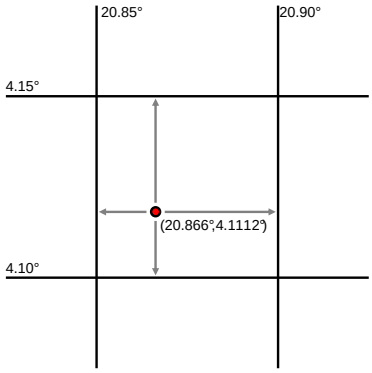

Figure 5.1: Generating query regions for the evaluation

catalogues and searched the SDSS catalogue for matches within a certain rectangular area around the objects from ROSAT or 2MASS. The resulting 730 entries have been used to generate the RA- and DEC-regions for our evaluation queries by rounding the RA- and DEC-coordinates off and up to the next multiple of 0.05 (three arc minutes) as shown in figure 5.1 and using the resulting RA- and DEC-intervals as query region. Listing 5.2 shows one of the queries used during the evaluation. We used the formulation with subqueries for each catalogue in order to force the optimizer to do an index-based scan on RA and DEC without using DBMS-specific optimization hints. For each of the nodes, we shuffled the 730 queries.

```
1 select * from (
2    select * from sdss
3       where ra between 112.15 and 112.20
4         and dec between 35.70 and 35.75
5 ) s1 left outer join (
```

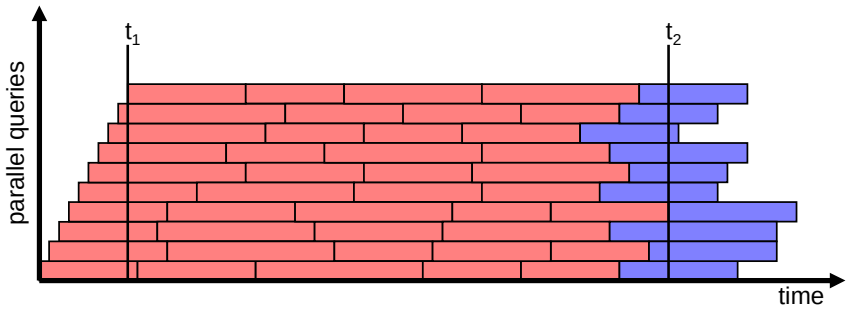

Figure 5.2: Processed queries at one peer within a specified time interval

```
6     select * from twomass
7        where ra between 112.15 and 112.20
8          and dec between 35.70 and 35.75
9  ) s2 on xmatch(s1, s2, 0.005 ) left outer join (
10    select * from rosat
11       where ra between 112.15 and 112.20
12         and dec between 35.70 and 35.75
13 ) s3 on xmatch(s1, s3, 0.005 )
```

Listing 5.2: HiSbase query expression which cross-matches three catalogues

We define the throughput as the number of queries returning results within a specified time span divided by the time span. In addition, HiSbase is able to process multiple queries at the same time, so we have to determine the throughput for different amounts of parallel queries — the so-called *multi-programming level*. An example with a maximum of ten parallel queries at one peer is shown in figure 5.2. The timestamp t_1 indicates the beginning of our measurement interval after the designated multi-programming level of ten has been reached and t_2 indicates the end of the interval when the multi-programming level is about to decrease because we have reached the end of the query batch. Each of the vertical bars represents a worker thread and the bar's segments represent queries. The length of each segment indicates the processing time of the related query. The red segments depict the queries which complete within our measurement interval while the blue ones will complete after its end.

We need to identify t_1 and t_2 individually for each tested multi-programming level based on the log files written by HiSbase during its operation. At first we can extract all timestamps of queries being injected at the local HiSbase node. For a multi-programming level of m we know that the multi-programming level will be reached after m injections, so we can skip the first $m-1$ timestamps to determine t_1. The decrease of the multi-programming level begins when the last query has been injected, so the last timestamp is t_2. Now we extract the timestamps of finished queries and count all timestamps from t_1 to t_2. This gives us the number n of results during the observed time interval. The throughput for the node is then $\frac{n}{t_2-t_1}$.

To calculate the throughput for the whole network of 16 nodes, we need to determine the latest t_1 and earliest t_2 of all nodes and then count the number of results within the interval from t_1 to t_2 on all nodes.

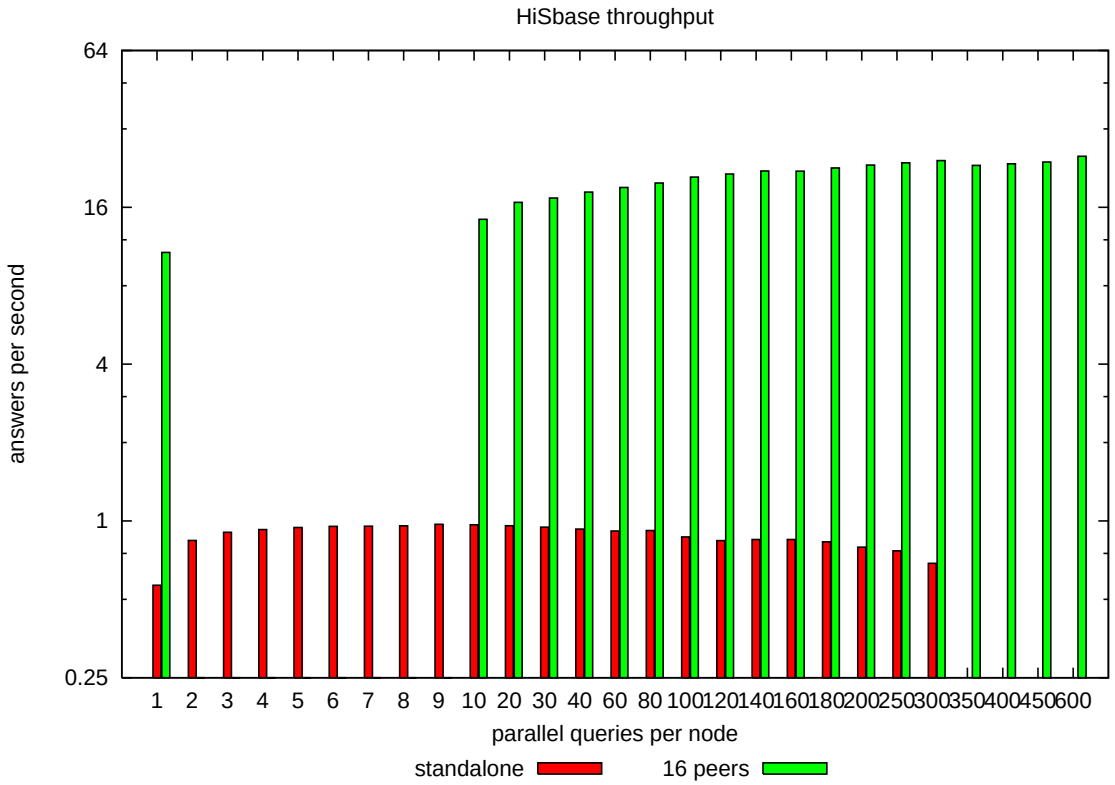

Figure 5.3: Throughput increase with HiSbase

5.2 Results

We ran our tests on the HiSbase network consisting of 16 peers (the green results in figure 5.3) with 730 queries and multi-programming levels of 1, 10, 20, 30, 40, 60, 100, 120, 140, 160, 180, and 200 at each node. In addition we ran tests which repeated the 730 queries once on each peer and hence injected 1,460 queries at each node for the multi-programming levels 250, 300, 350, 400, 450, and 600.

On the standalone HiSbase node we ran tests with 730 queries and multi-programming levels of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 60, 80, 100, 120, 140, 160, 180, and 200 (the red results in figure 5.3).

It is easy to see that the maximum throughput of the HiSbase network with 16 nodes is far more than 16 times the throughput of a standalone HiSbase node which indicates a super-linear speedup. In addition, the maximum throughput on a standalone node is reached with a multi-programming level between 5 and 10 while throughput of 25.13 results per second on the network with 16 nodes is reached with a multi-programming level of about 600 on each peer hence about 9,600 for the whole network. Since we did not accomplish tests with multi-programming levels above 600, we cannot rule out that a still higher throughput can be achieved on the network with 16 nodes.

Chapter 6

Conclusion

During this work we have discussed the development of the HiSbase prototype allowing storage, processing, and retrieval of locality-aware data in a distributed peer-to-peer environment. This gives scientists the ability to form so-called *virtual organisations* in which each participant provides its own computing and storage resources and can use the resources made available by the other participants.

The astrophysical catalogues for which HiSbase has been initially designed contain highly skewed data. HiSbase handles this skew by using a subset of the data to create a histogram during a training session. An abstract interface allows us to support different histogram types, e. g. equi-depth histograms or Z-quadtrees. Instead of individual data objects, every histogram region including all its data is mapped onto the key space of the underlying peer-to-peer system FreePastry [FP07] to preserve locality.

Scientists can query the distributed catalogues using a SQL-based syntax to formulate queries at any participating node. After preparing the issued query, all regions covered by the query are determined, a coordinating node is selected, and the query is sent to the coordinator which splits up the query and sends it to all covered regions. Each node maintaining at least one of the covered regions queries its local database and sends the results back to the coordinator which combines all results and sends them to the node where the query has been injected.

In astrophysics, cross-correlating data from different catalogues is an important operation. HiSbase allows scientists to combine the data from different catalogues by applying a cross-match predicate to a query. This cross-match predicate offers the possibility of joining two catalogues and retrieving only those data objects from the second one which are within a certain neighbourhood to data objects from the first catalogue. Using multiple cross-matches, more than two catalogues can be combined. The possibility of correlating data from several large catalogues features an instrument which allows scientists to obtain new results based on existing information.

HiSbase kind of sorts the injected queries by distributing them among the individual nodes holding the requested data. This allows to preserve locality and to process the queries in parallel instead of sequentially. This results in an increased need for computing equipment which can be satisfied using already installed systems on the one hand but allows a much higher throughput with fairly moderate technical requirements as shown in our evaluation in chapter 5 on the other hand.

Although the intended application of HiSbase is supporting astrophysicists in accessing and combining data from different astrophysical catalogues, it is possible to utilize HiSbase for other scientific fields, e. g. meteorology or geography. The wide applicability of HiSbase shows that computer science has become a key discipline supporting other scientific fields.

Chapter 7

Future work

Now that we have done a first step evaluating our prototype, it is time to look ahead at the next possible steps in its development. Some of the upcoming points are already planned, others are just ideas about new features that could be needful for the e-science community.

7.1 Median-Z-quadtrees

As already mentioned in the previous chapters, our first approach to partitioning the spatial data used the Z-curve. This allowed us to get equi-depth histograms out of a representative subset of the spatial data. The geometric overhead going hand in hand with the Z-curve histograms lead us to Z-quadtrees which allowed us much easier handling due to their regular shapes. Unfortunately with Z-quadtree histograms compared to the Z-curve the filling level of each bucket will differ to a greater extent from the other buckets, as shown in figure 7.1. In addition, data concentrating in one or few "hot spots" can lead to Z-quadtrees with one branch going deeper and deeper, as seen in figure 7.2 on the next page. This increases the cost of looking up the regions affected by a query.

As a possible solution we figured out the concept of Median-Z-quadtrees where we quarter a region no longer at its geometric centre but at a point defined by the median on each dimension. Using this technique, the growth of the tree can focus faster on highly populated regions and therefore reduce the overall depth of the resulting tree, as shown in figure 7.3 on the following page.

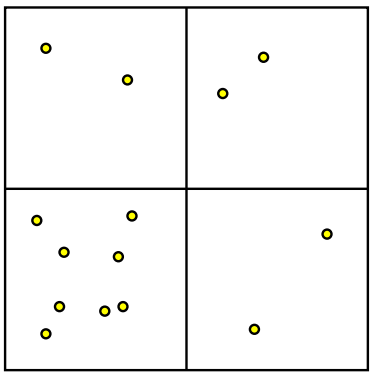

Figure 7.1: Spatial area, organized as Z-quadtree, split after more than 9 points in one bucket

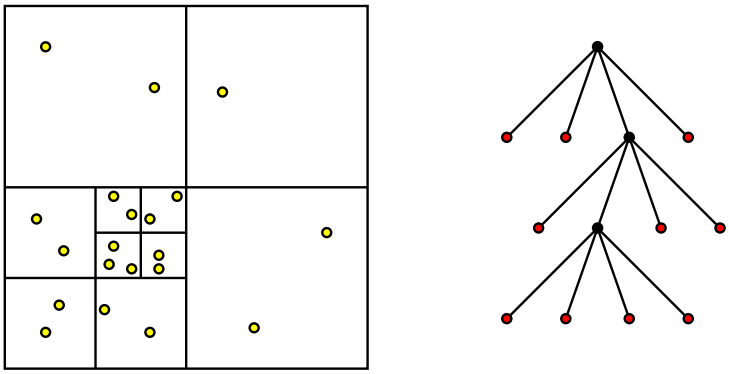

Figure 7.2: Spatial area with a local density and a resulting Z-quadtree

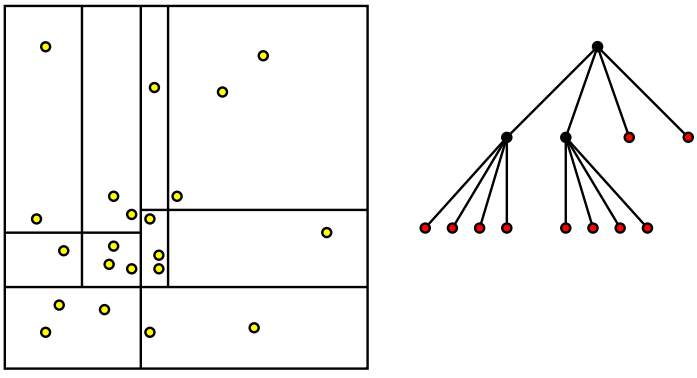

Figure 7.3: Spatial area with a local density and a resulting Median-Z-quadtree

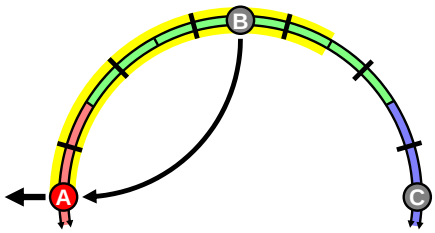

Figure 7.4: HiSbase network processing a query which covers the yellow marked regions. Node A acts as coordinator, B has to send its results to A.

7.2 Alternative coordinator selection

With a growing number of participating nodes in the HiSbase network and growing physical distances between the nodes, a deliberate selection of the coordinator will get more important. Depending on this selection, network load in the sense of partial result sets being routed between the nodes can be saved or increased, which can directly influence the network's performance and the operating cost.

Our current approach in selecting the coordinator depends on the regions affected by a query. The node at which the query is injected into the network calculates the affected regions. If it holds one of the affected regions itself, it will become the coordinator. This already helps reducing network load, because it is likely that a user injects the query into the network at a node which is close to him. If the node does not hold any of the affected regions, it will choose the node holding the first affected region as coordinator as shown in figure 7.4. In some case, this can increase network load.

The alternative approach will not choose the node holding the first affected region, but the node holding the middle region (see figure 7.5). In this case, it is very likely that the chosen node not only holds the middle region, but also several regions above and below. This increases the amount of regions queried at the coordinator and decreases the amount of regions queried at other nodes. Therefore the size of the answers from the other nodes to the coordinator and consequently the network usage are reduced.

7.3 Alternative staging techniques

Due to the focus on query processing in this thesis and the main goal of developing a HiSbase prototype for first presentations, we implemented a rather simple staging mechanism. Each node first cleans up its local database to remove all entries originating from a previous run. The node then uses the histogram to calculate the spatial areas for which it will be responsible. After this calculation is done, the node queries a database archive for those spatial areas and inserts them one after another into its local database.

For adapting to later changes in the HiSbase network caused by leaving or joining nodes, this procedure is automatically repeated when a node observes changes within its direct

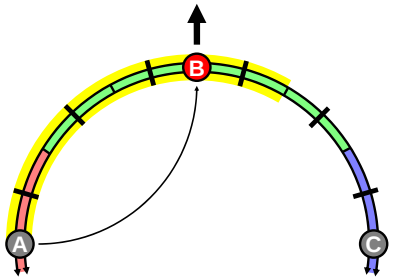

Figure 7.5: The same situation but now node B acts as coordinator and the data being delivered from A to B is now clearly reduced.

neighbourhood. To prevent these automatic updates from occurring too often during network instabilities, they will not be executed directly after the observed changed but with a specific delay. This delay allows an unstable network to "calm down" before the local data is updated.

Obviously, this staging mechanism is not adequate for a big scenario with possibly thousands of peers. One or several central archives would be overwhelmed with all the parallel requests from the nodes updating their data. To resolve this problem, we considered different approaches using advanced or streaming techniques.

7.3.1 Advanced data staging

The currently implemented data staging can be extended to reduce load on the central archives and distribute the overhead of staging on other nodes. When an additional node enters the HiSbase network, it can contact both neighbouring nodes, request their local database settings, and utilize their databases as archive. After retrieving the data from its neighbours, the node has to signal them that they may now "forget" those spatial areas.

Another advantage can be implemented when a node leaves the HiSbase network. Instead of restaging the whole data, both nodes neighbouring the leaving node can determine the newly gained regions and only fetch their data from the central archives.

7.3.2 Streaming the data

The streaming approach assumes that there are several archives, each of them storing the data of one spatial catalogue. Some of the archives might be databases allowing SQL-queries while others might just be CSV (comma-separated values) files. A HiSbase node located close to an archive can access the archive and send the data row by row into the HiSbase network in regular intervals. The already existing routing mechanism of HiSbase can be adapted to ensure that each data row gets routed to the node holding the related spatial area. This technique would guarantee nodes getting updated information from time to time without overwhelming one or more archives but new nodes joining the network for the first time might have to wait some time until they are properly equipped with their data.

Another streaming approach can be used as an enhancement to speed up delivery of data to new nodes. Existing nodes already monitor the underlying Pastry layer for changes in their direct neighbourhood. These changes can result from new nodes joining the network or nodes leaving the network or experiencing failures. In the first case, the new node joining the network will inherit some of the regions of its direct neighbours. Both nodes directly neighbouring the new node can observe the change in their neighbourhood and calculate the regions they have "lost" to the new node. All data rows belonging to these regions can then be streamed to the new node using the technique described above. This allows a HiSbase network to handle joins without having to wait for the archives to republish their data. Only the case when nodes leave the network can not be handled easily.

7.4 Exchanging histograms between participating nodes

Currently every HiSbase node participating in a network has to have the histogram file to correctly initialize its own data and route locally injected queries to the coordinator. The prior manual replication of the histogram file can be time-consuming and vulnerable to mistakes. In addition, discrepancies between histogram files can cause consistency problems when queries are injected at the node using the faulty histogram.

A possible solution would be that only the first node uses a local histogram file and the other nodes acquire a copy from its bootstrap node or its direct neighbours. This eliminates the necessity of distributing the histogram file to all nodes before launching the network and — as a spin-off — removes the associated consistency problems because all nodes automatically use the same histogram file. The other configuration parameters of a HiSbase network can also be exchanged during the bootstrap procedure to a limited extent.

7.5 Connecting clients

Although being desirable not all potential users of the HiSbase system will be able to share their storage and computing resources. Users might have limitations on their resources or experience security restrictions at their institutes. Consequently it will not be possible for these users to take part in the system as a regular HiSbase node. In this case, we will need a possibility of injecting queries into a network and retrieving the results using one of the regular nodes.

This could be done by offering a secured connection to a HiSbase node including the necessary authentication. Using this connection, scientists can inject queries at the HiSbase node and wait for the results as shown in figure 7.6. Moreover, this could lead to a stricter separation between well equipped HiSbase servers organized in a peer-to-peer network doing the query processing and regular office computers allowing users to access the data. Furthermore the HiSbase servers could buffer incoming results of an issued query until the requesting scientist contacts the server again and retrieves the result.

7.6 Configuration of output formats and files

The current implementation returns results of queries in a simple ASCII presentation as shown in listing 7.1. Although being human readable, such a format is not very suitable

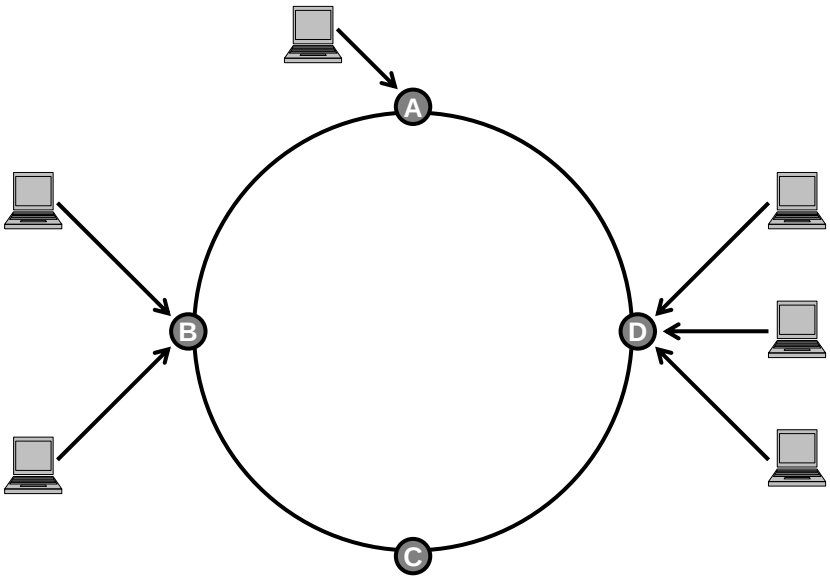

Figure 7.6: Several clients connecting to a HiSbase network with four nodes

for further processing. To allow the further processing of data retrieved from a HiSbase network, it would be desirable to write the results into files and convert them into more sophisticated formats like XML or CSV.

```
RA                  DEC                   OBJID
-------------------------------------------------------
43.0028805407238   0.0532208965953161   582104534777266605
43.0073125412446   0.0571183304587983   582104534777267027
43.0105129536098   0.074731940838761    582104534777266433
43.0105129559039   0.0747319310339798   582104534777266432
43.0121250100818   0.070822639284091    582104534777266869
43.012647146706    0.0575093823663683   582104534777266627
43.0137587716116   0.05257379681051     582104534777267031
43.014699813568    0.054560408797727    582104534777266630
43.0155633938737   0.0551062842858088   582104534777266631
43.0155635377075   0.0551056695489369   582104534777266629
43.0183998713865   0.0554794262627254   582104534777267034
43.0190408732614   0.062027069640759    582104534777267035
43.0197552285185   0.0691341278595079   582104534777267037
43.0207281633403   0.066583239237338    582104534777267038
43.0240778701781   0.0579166456854712   582104534777266645
43.0241068185943   0.0596235445365354   582104534777266643
43.0245649485094   0.0731968795573173   582104534777266883
```

Listing 7.1: Raw ASCII output of a query's result

Alternative output formats could be implemented as filters which transform the results into the desired format. Using a pluggable architecture for those filters, each scientific domain, institute, or even user could supply and use user-defined filters.

7.7 Handling query hot spots

In addition to data skew, a HiSbase network might also experience query hot spots when a huge number of queries targets the same spatial region and is therefore routed to the same HiSbase node. A simple solution to this problem would be to record issued queries over some time and use them for later refinement of the applied histograms. The queries could be used by the histogram trainer just as any other spatial object and hence allow the trainer to separate a bigger region which experiences query hot spots into several smaller regions. This can be easily implemented but does not offer any load balancing. It only reduces the probability of receiving queries apart from the hot spot on a node suffering from a hot spot.

Another solution offering limited load balancing would be to position multiple database servers at each HiSbase node suffering from query hot spots. This would not improve locality because these multiple database servers do not share their main memory, but it would improve parallelism during processing. More possible solutions with multiple HiSbase nodes being responsible for the intensively queried regions would require extensions of the underlying peer-to-peer network, allowing the same keys — for the same regions — to be stored more than once.

Bibliography

[ADD07] Apache Derby, February 2007. `http://db.apache.org/derby/`.

[Bau07] Bernhard Bauer. Locality-aware Histogram Structures for P2P Data Management. Systementwicklungsprojekt, Technische Universität München, January 2007.

[Cla99] Ian Clarke. A Distributed Decentralised Information Storage and Retrieval System, 1999.

[DB207] IBM DB2, February 2007. `http://www-306.ibm.com/software/data/db2/`.

[DZD+03] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a Common API for Structured Peer-to-Peer Overlays, 2003.

[EDB06] *Proc. of the Intl. Conf. on Extending Database Technology*, Munich, Germany, March 2006.

[EJ101] Donald E. Eastlake 3rd and Randy Jones. RfC 3174: US Secure Hash Algorithm 1 (SHA1), September 2001. `http://www.ietf.org/rfc/rfc3174.txt`.

[FB74] Raphael A. Finkel and Jon Louis Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.*, 4:1–9, 1974.

[Fos02] Ian Foster. What is the Grid? A Three Point Checklist. *GRID Today*, July 2002. `http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf`.

[FP07] FreePastry, February 2007. `http://www.freepastry.org/FreePastry/`.

[GDF07] Gnutella Protocol Development, February 2007. `http://www.the-gdf.org/`.

[GNSS06] Jim Gray, María A. Nieto-Santisteban, and Alexander S. Szalay. The Zones Algorithm for Finding Points-Near-a-Point or Cross-Matching Spatial Datasets. Technical report, Microsoft Research, April 2006. MSR TR 2006 52.

[Hil91] David Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Math. Ann.*, 38:459–460, 1891.

[HSQ07] HSQLDB, February 2007. `http://hsqldb.org/`.

[KAZ07] KaZaA, February 2007. `http://www.kazaa.com/`.

[PNT06] T. Pitoura, N. Ntarmos, and P. Triantafillou. Replication, Load Balancing, and Efficient Range Query Processing in DHT Data Networks. In EDBT06 [EDB06], pages 131–148.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 161–172, August 2001.

[Sag94] Hans Sagan. *Space-Filling Curves*. Spinger-Verlag, 1994.

[SBK+06] Tobias Scholl, Bernhard Bauer, Richard Kuntschke, Daniel Weber, Angelika Reiser, and Alfons Kemper. HiSbase: Histogram-based P2P Main Memory Data Managment. November 2006.

[Sch05] Tobias Scholl. Distributed Processing of Data Streams in P2P Networks. Diploma thesis, Universität Passau, September 2005.

[SCS+06] M.F. Skrutskie, R.M. Cutri, R. Stiening, M.D. Weinberg, S. Schneider, J.M. Carpenter, C. Beichman, R. Capps, T. Chester, J. Elias, J. Huchra, J. Liebert, C. Lonsdale, D.G. Monet, S. Price, P. Seitzer, T. Jarrett, J.D. Kirkpatrick, J. Gizis, E. Howard, T. Evans, J. Fowler, L. Fullmer, R. Hurt, R. Light, E.L. Kopan, K.A. Marsh, H.L. McCallon, R. Tam, S. Van Dyk, and S. Wheelock. The Two Micron All Sky Survey (2MASS). *Astrophysical Journal*, 131(1163), 2006.

[SDS07] The SDSS Data Release 5, February 2007. `http://www.sdss.org/dr5/`.

[Set07] Sanjeev Setia. Distributed Hash Tables (DHTs): Tapestry & Pastry, February 2007. `http://cs.gmu.edu/~setia/cs699/lecture2.pdf`.

[SJ07] Java, February 2007. `http://java.sun.com/`.

[Sky07] SkyServer, February 2007. `http://skyserver.sdss.org/`.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, August 2001.

[USN07] The USNO-B1.0 survey, February 2007. `http://www.nofs.navy.mil/data/fchpix/cfra.html`.

[VAB+99] W. Voges, B. Aschenbach, T. Boller, H. Bräuninger, U. Briel, W. Burkert, K. Dennerl, J. Englhauser, R. Gruber, F. Haberl, G. Hartner, G. Hasinger, M. Kürster, E. Pfeffermann, W. Pietsch, P. Predehl, C. Rosso, J. H. M. M. Schmitt, J. Trümper, and H. U. Zimmermann. The ROSAT all-sky survey bright source catalogue. *Astronomy and Astrophysics*, 349:389–405, September 1999.

[ZHS+04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. In *IEEE Journal on Selected Areas in Communications (JSAC), Vol 22, No 1*, pages 41–53, January 2004.

Acknowledgments

I would like to deeply thank everyone who provided me with advices and helped me in any way during the time I spent on this work, especially the following people:

Tobias Scholl, for leading the HiSbase project, choosing me as a participant, guiding me through this thesis, and helping me to defeat Java.

Angelika Reiser, co-leader of the HiSbase team, for her guidance through almost three years of my studies.

Bernhard Bauer, for his unorthodox but excellent ideas during his participation in the HiSbase project.

Prof. Alfons Kemper, Ph.D., for giving me the chance of participating in the HiSbase team and getting in touch with other scientific topics.

Jeff Hoye, for his excellent support of FreePastry [FP07] which helped us to save hours of debugging.

Benjamin Gufler, for being an always helping friend during my studies and for sharing his Java knowledge.

Monika Toth, for convincing me to attend a university and for her proofreading and linguistic advice.