Issues on Distributed Caching of Spatial Data

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Carlos Georg Lübbe

aus Bühl

Hauptberichter: Prof. Dr.-Ing. habil. Bernhard Mitschang
Mitberichter: Prof. Mario A. Nascimento
Tag der mündlichen Prüfung: 24. Januar 2017

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2017

Contents

List of Acronyms **7**

Abstract **9**

Deutsche Zusammenfassung **11**

1 Introduction **13**

1.1 Field of Application 14
1.2 Requirements for the Data Management 17
1.2.1 Availability 17
1.2.2 Transparency 18
1.2.3 Spatial Support 19
1.2.4 Locality Exploitation 19
1.3 State of the Art of the Technology and Research 19
1.4 Contributions 22
1.5 Outline of this Thesis 26

2 **Design Considerations for a Distributed Caching Architecture** **29**
2.1 System Environment 30
2.1.1 Data Representation and Back-end Technologies . . . 31
2.1.2 Query Representation and Evaluation Strategy 32
2.1.3 Data Maintenance 34
2.2 Designing a Cache Server 35
2.2.1 Operation Mode 36
2.2.2 Data Granularity for the Cache Management 37
2.2.3 Search Mechanisms 43
2.2.4 Caching Strategies 45
2.2.5 Cache Memory Types 46
2.3 Designing a Cluster of Multiple Cache Servers 48
2.3.1 Main Tasks during Operation 48
2.3.2 Degree of Distribution 49
2.3.3 Access Mechanisms for the Cluster 52
2.3.4 Internal Network Design of the Cluster 54
2.4 Proposed Software Architecture 56
2.5 Summary and Outlook 58

3 **A Cache Layer Implementation for Spatial Data** **59**
3.1 Spatial Partitions as Basic Units for Caching 60
3.2 Operation Mode 62
3.3 Caching Strategies 63
3.3.1 Traditional Caching Strategies 63
3.3.2 Focused Caching 63
3.4 Assessment 65
3.4.1 Space Complexity 65
3.4.2 Time Complexity 67
3.4.3 Data Identification Accuracy 68
3.4.4 Summary 70
3.5 Beyond Spatial: Supporting Alternative Attribute Dimensions 71
3.5.1 Type-enabled Parcel Caching 71
3.5.2 Discussion 72
3.6 Summary and Outlook 73

4 Implementation Concepts for the Cluster Layer **75**
4.1 Partition-based Content Arrangement 76
4.1.1 A Parcel-based Cluster Layer Implementation 76
4.1.2 Discussion 89
4.2 Content Arrangement Through Focused Caching 91
4.2.1 A Particle-Spring-based Implementation 93
4.2.2 A Delaunay-based Implementation 111
4.2.3 Observations on Scalability 130
4.3 Discussion 132
4.4 Summary and Outlook 133

5 Assessment Framework **135**
5.1 A Framework for Visual Example Applications 136
5.2 A Simulation Framework 139
5.3 An Evaluation Framework for Cluster Environments 142
5.4 Summary and Outlook 143

6 Conclusion **145**
6.1 Review 145
6.2 Practical Use 146
6.3 Future Prospects 147

List of Figures **149**

List of Tables **151**

List of Listings **153**

Bibliography **155**

List of Acronyms

ACID Atomicity, Consistency, Isolation and Durability

BASE Basically Available, Soft State and Eventually Consistent

CAN Content Addressable Network

CAP Consistency, Availability and Partition Tolerance

CPU Central Processing Unit

DBS Database System

DNS Domain Name System

GPS Global Positioning System

IP Internet Protocol

LBS Location-based Service

OSM OpenStreetMap

PDF Probability Density Function

POI Point of Interest

SQL Structured Query Language

WKT Well-known Text

Abstract

The range of digital information on places or locations has been growing rapidly in recent times. Along with the spreading of Internet-ready mobile devices allowing access to such information anytime and anywhere, location-based services and applications have gained enormous popularity. Both, the expanding user base of such services as well as the ever growing data volumes in that sector impose serious challenges on the providers of spatial data. For instance, the data provisioning process must be fairly efficient in order to achieve cost-effective operation, the resources must be allocated flexible enough to compensate dynamic load peaks during runtime, and location-based services must be able to scale with growing data volumes and workloads.

In this work, we propose a distributed spatial cache between the data provider and the clients. In the distributed spatial cache, multiple independent cache servers store a copy of the most frequently used data in memory. With our distributed caching approach, we address the challenges of spatial data providers as follows: First, we devise a dedicated caching strategy

which considers the spatial data access patterns of location-based applications. Thus, the overall efficiency is increased, as a great portion of the previous query results can be reused to serve the requests. Second, resource allocation is extremely flexible in the cache, as the cached data is only a replica which can be dropped at any time without data loss. This enables us to design advanced load-balancing mechanisms which compensate dynamic load shifts during run-time. And third, we define protocols for adding and removing cache servers to or from the distributed spatial cache. Thus, the capacity can be seamlessly scaled with increasing or decreasing demands.

In this thesis, we examine the characteristics and requirements of data provisioning in the context of location-based services. Then, we derive possible design strategies and introduce a general software architecture for the distributed spatial cache. We present several concrete implementation approaches and compare them. Our evaluation shows not only the principal feasibility of caching in the context of location-based applications, but also the effectiveness of caching for achieving scalable and adaptive system designs in the context of provisioning of spatial data.

Deutsche Zusammenfassung[1]

Die Menge an digitalen Informationen über Orte hat bis heute rapide zugenommen. Mit der Verbreitung mobiler, internetfähiger Geräte kann nun jederzeit und von überall auf diese Informationen zugegriffen werden. Im Zuge dieser Entwicklung wurden zahlreiche ortsbasierte Anwendungen und Dienste populär. So reihen sich digitale Einkaufsassistenten und Touristeninformationsdienste sowie geosoziale Anwendungen wie Foursquare in der Liste der beliebtesten Vertreter. Steigende Benutzerzahlen sowie die rapide wachsenden Datenmengen, stellen ernstzunehmende Herausforderungen für die Anbieter ortsbezogener Informationen dar. So muss der Datenbereitstellungsprozess effizient gestaltet sein, um einen kosteneffizienten Betrieb zu ermöglichen. Darüber hinaus sollten Ressourcen flexibel genug zugeordnet werden können, um Lastungleichgewichte zwischen Systemkomponenten ausgleichen zu können. Außerdem müssen Datenanbieter in der Lage sein, die Verarbeitungskapazitäten mit steigender und fallender Anfragelast zu skalieren.

[1] A summary of this thesis in german.

Mit dieser Arbeit stellen wir einen verteilten Zwischenspeicher für ortsbasierte Daten vor. In dem verteilten Zwischenspeicher werden Replika der am häufigsten verwendeten Daten von mehreren unabhängigen Servern im flüchtigen Speicher vorgehalten. Mit unserem Ansatz können die Herausforderungen für Anbieter ortsbezogener Informationen wie folgt addressiert werden: Zunächst sorgt eine speziell für die Zugriffsmuster ortsbezogener Anwendungen konzipierte Zwischenspreicherungsstragie für eine Erhöhung der Gesamteffizienz, da eine erhebliche Menge der zwischengespeicherten Ergebnisse vorheriger Anfragen wiederverwendet werden kann. Darüber hinaus bewirken unsere speziell für den Geo-Kontext entwickelten Lastbalancierungsverfahren den Ausgleich dynamischer Lastungleichgewichte. Letztlich befähigen unsere verteilten Protokolle zur Hinzu- und Wegnahme von Servern die Anbieter ortsbezogener Informationen, die Verarbeitungskapazität steigender oder fallender Anfragelast anzupassen.

In diesem Dokument untersuchen wir zunächst die Anforderungen der Datenbereitstellung im Kontext von ortsbasierten Anwendungen. Anschließend diskutieren wir mögliche Entwurfsmuster und leiten eine Architektur für einen verteilten Zwischenspeicher ab. Im Verlauf dieser Arbeit, entstanden mehrere konkrete Implementierungsvarianten, die wir in diesem Dokument vorstellen und miteinander vergleichen. Unsere Evaluation zeigt nicht nur die prinzipielle Machbarkeit, sondern auch die Effektivität von unserem Caching-Ansatz für die Erreichung von Skalierbarkeit und Verfügbarkeit im Kontext der Bereitstellung von ortsbasierten Daten.

CHAPTER 1

Introduction

Location-based services and applications have gained enormous popularity in recent times. This development is due to the ever increasing proliferation of Internet-ready mobile devices allowing access to digital information anytime and anywhere, as well as the fast-growing range of digitally available information on places or locations. Both the growing data volumes and the expanding user base impose serious challenges on providers of such spatial information: For instance, systems must be able to scale with ever growing data volumes and workloads, the resources must be allocated wisely and flexible enough to compensate dynamic load peaks during the execution and the system architecture must be resistant to failures of system components to ensure the availability of the service. This work proposes a caching framework providing highly available access to the most frequently used spatial data and thus can constitute an essential ingredient for data providers in addressing these challenges.

In the following, we use a concrete application example to characterize the application field. From this example, we derive specific challenges and requirements for the data processing in this context. Subsequently, we discuss to which extent current technology meets the requirements and outline the gap between these requirements and technologies. Finally, we describe how this work contributes to bridging this gap.

1.1 Field of Application

The main objective of many location-based applications is to extract carefully chosen data from a vast amount of data sources and present it to the user in a meaningful way. On mobile devices, spatial information applications showing the user interesting places surrounding his current position have received much attention recently. Known examples range from digital shopping assistants and tourist guides to geosocial apps, such as Foursquare [Foursquare, 2013]. To be more specific, we give an exemplary task description in natural language which a typical spatial information application could possibly carry out:

Present the user a live view which visualizes shops and point of interests surrounding his current position superimposed on a map of the city.

The following characteristics can be derived from this example task:

- **Various data sources:** The task description involves diverse data sources. For instance, the request to draw a map includes geometries of buildings, line strings of streets, points of interest, textual descriptions such as street labels, pictures of sites which were taken at a certain position or even complex 3-d models of certain real world objects. A common property shared by these different data objects is that each of them can be mapped to a real world position and that they are fairly static in a sense that these data objects are rarely updated. Today such data is generally available and typically stored in huge geographic databases such as GoogleMaps[1] or OpenStreetMap[1].

[1] The services can be accessed under http://www.maps.google.com and http://www.openstreetmap.org, report status: Apr. 17, 2017

In addition to these fairly static information sources, the example includes dynamic information describing the user's current context which can be easily derived from sensors of the user's mobile device. In our example the user's context is his current location which can be derived from the GPS sensor or by Wi-Fi positioning.

- **Location as primary selection criterion:** Of course any information source could be included to describe the user's context. However, according to [Yoo, 2007] a broad range of context-aware applications regard location information as primary context, i.e., they use the location information to retrieve or select data which is relevant to the user and then in a second step further refine or adjust their selection using secondary context information. The spatial information application for example uses the location of the user as primary context to retrieve map data surrounding the user's current position. In a second step, the application could use the current time and the opening times of the shops to mark their opening status with a dedicated color in the visualization. The focusing on location as primary context may lead to an invalidation of the data extracted during the selection step as soon as the location changes. To keep the presented information relevant to the user, the extracted data must be updated as soon as the context changes. This leads to frequent data requests and thus generates a lot of workload for the providers of spatial information. For instance, our example visualization process will generate a new request for shops and POIs in the surrounding regions each time the GPS sensor of the user's device reports a new position.
- **Spatial locality in data access patterns:** Succeeding requests of a single client can overlap to a great extent. In our example this happens whenever the map of the user's current surroundings is updated and the user's new position is close to the old position. We denote this phenomenon as *intra client locality*. In addition to that, the requests of different clients can exhibit large spatial locality, which is denoted as *inter client locality* in the following. Inter client locality occurs when two different clients request the same or similar data. In our example this could happen when two persons use their spatial

information application at the same or similar positions. As certain behavioral patterns of users recur regularly (e.g., searching for hotels nearby after arrival at the main train station of a city), inter client locality is very common in our application field.

- **Reduced consistency:** Nowadays spatial data is freely accessible over the Internet. Yet, spatial data is often incomplete, inconsistent, imprecise and inaccurate as stated by [Chrisman, 1991]. In fact, it is widely agreed that a perfect model of the world cannot be obtained as the described real world entities continuously evolve or change, e.g., roads are being built, city districts developed and even whole continents are slowly drifting apart. As it is difficult to keep track of such real-world changes, spatial data tends to become inaccurate or even inconsistent in practice. The inherent inconsistency of spatial data can be approached with two attitudes: First, explicitly model the quality of data and thus provide applications the means to measure quality of provided data – an approach we have followed in a preliminary work as described in [Grossmann et al., 2009]. Or second, shift the problem to the users and let them cope with the inconsistencies. The currently favored approach seems to be the second one, as most location-based applications silently accept inconsistencies and present them to the user in an unfiltered way. Occasionally, this includes car navigation services sending their customers onto airplane taxiways of airports as reported by [Cole, 2013]. In addition to the inherent inconsistency of spatial data, inconsistencies can occur when data is stored in a distributed system. In this context, it makes sense to mention Eric Brewer's CAP theorem as described in [Gilbert and Lynch, 2002]. It states that a distributed system cannot provide all three of the following guarantees at the same time: *consistency* (i.e. concurrent operations see the same data), *availability* (i.e. each request receives a response) and *partition tolerance* (i.e. the system continues operation despite partial failures). While traditionally data providers focused on consistency, nowadays consistency is often dispensed for the sake of high performance and scalability. For this reason, the ACID paradigm (**A**tomicity, **C**onsistency, **I**solation and **D**urability) does not

work anymore to describe the transactional properties of such data offerings. Instead the acronym BASE (**B**asically **A**vailable, **S**oft-state and **E**ventually consistent) is suggested to characterize systems with reduced consistency support [Pritchett, 2008]. As applications need to be able to cope with the inherent inconsistencies of spatial data in any way, it is questionable whether the enhanced transactional consistency of ACID brings much benefit. For this reason, we believe that BASE's softened requirements for transactional consistency fit well with our application field. This creates opportunities to implement the crucial requirements for the data management which are presented in the following section.

1.2 Requirements for the Data Management

From the characteristics of the application field discussed in the previous section, the following main requirements for the data management can be derived:

1.2.1 Availability

To achieve high availability of a system, three main ingredients are mandatory: scalability, adaptivity and fault tolerance. In the following we detail on each of these aspects.

- **Scalability:** A rapidly growing number of people using location-based services or applications impose an enormous amount of spatial requests on the data tier of such a service. The ability to scale a system's capacity smoothly with an increasing number of users is vital for a service provider. If this requirement is not met, overload might drive the response times to the top or even cause a total crash of the service – a circumstance, which massively affects the quality of service perceived by the users. In the worst case users might refuse to use the service any longer and thus may cause an otherwise successful business idea to fail. An additional argument for a scalable system design

is the increase of cost efficiency driven by the economies of scale, i.e., it allows supplying a larger user base without further development costs.

- **Adaptivity:** Using a mobile device, people typically access location-based services during their daily life. In consequence, the expected workloads will correspond to the users' current activity and may severely change over the time. For this reason, the data management must be flexible enough to cope with dynamic load peaks and shifting data access patterns.
- **Fault tolerance:** System failures causing unresponsive services are annoying for the user, especially when the service is needed most. This is particularly true for location-based services, as these are often accessed by users on the go when a quick and reliable response is required. Before entering a subway, for instance, a user has only seconds to find out whether the arriving line will bring him/her to his/her desired destination. Therefore fault tolerance is a key requirement in our application field. In consequence, data must be delivered to the end-users even in the event of a failure of system components.

1.2.2 Transparency

Modern system architectures rely on loosely coupled system components, i.e., the data management tier is typically strictly separated from application or presentation logic. This simplifies the development, increases maintainability of code and brings in flexibility in terms of interchangeability of system components. To reach this goal, all tasks specific to the data management must happen completely transparent to the other system components. This is particularly true for the implementation of the above mentioned requirements. I.e., neither scale-in, scale-out, the failure of single system components nor any adaptation activities should affect the application logic.

1.2.3 Spatial Support

As most location-based applications use spatial context as primary selection criterion for retrieving relevant information, spatial support should be treated as a first class citizen by data providers. The easiest and at the same time most important query which includes location is the region query, also known as spatial window query. In this type of query the user wants to find objects that spatially interact (intersection, containment, contact etc.) with a given area of interest. It turns out that the mechanisms required to process region queries efficiently can also be exploited to process more complex spatial query types. For instance, to retrieve the k nearest objects to a given location, a query processor can successively issue range queries to expand the search space around the search location until the k nearest objects are found, as described by [Schwarz et al., 2004]. For this reason, we regard efficient range query processing as a fundamental ability in our target application field and therefore address our main attention to this type of query.

1.2.4 Locality Exploitation

As a lot of spatial and temporal locality in the data access patterns is to be expected in our target application field, we expect the ability to exploit such locality to be very beneficial for efficient resource utilization. A system that is unaware of access locality is in danger of unnecessarily wasting resources by making data available which is currently not requested at all. It is our expectation that the awareness of data access locality can significantly reduce the total resource consumption of data management tasks, as the resources can then specifically be devoted to the most frequent data access patterns.

1.3 State of the Art of the Technology and Research

After having described necessary requirements for the data management in our application field, this section focuses on the state of the art of related technology and research. In the following, we examine several spe-

cific technologies in order to discern to which extent they can meet the requirements.

- **Client-side caching:** With a client-side cache storing the most frequently used data of a single client as proposed by [Dar et al., 1996], the client application doesn't need to contact the data back-end whenever the requested data is already cached by this client. Thus, this approach exploits intra client locality. To maintain transparency, the client-side cache can be installed as a driver intercepting and interpreting the data requests of the client application. In the same manner, it is also possible to include spatial processing capabilities. However, client-side caching is of no use when multiple clients request similar data as the clients do not share their caches amongst each other. Moreover, client caches such as [Dar et al., 1996] focus on temporal locality in data access patterns and neglect spatial locality. In our application field many users may use their spatial information application in central places of the city such as main hall, main station or the airport. In this case the data back-end has to process the same or similar queries again and again. This causes high load on the network and especially at the data back-end which decreases the performance and possibly causes unavailability of the system.
- **Single server database systems (Single DBS):** When multiple clients request similar data, we say their data access behavior possesses *inter client locality*. To exploit this locality type, server-side caches are required. For common database management systems a dedicated buffer manager accomplishes this task by keeping the most frequently used pages in main memory. However, common buffer managers typically focus on temporal locality and pay no attention to spatial locality. Even though the database buffer usually occupies the lion's share of the database server's main memory in standalone client-server architectures and thus can easily reach multiple terabytes for modern hardware architectures, it can become a performance bottleneck as the buffer size is constrained by the server's hardware which cannot flexibly scale with an increasing number of clients. For this reason the availability cannot be ensured for an arbi-

trary number of clients. In a database system, transparency towards the application is achieved through a descriptive query language. Typically, such systems can be upgraded with spatial extenders, enabling enhanced spatial processing capabilities.

- **Parallel database systems (Parallel DBS):** In the field of parallel database processing high availability is achieved through hardware redundancy at several levels of the system. The three general architectures *shared memory, shared disk* and *shared nothing* are most prominent [Bhide, 1988, DeWitt and Gray, 1992, Stonebraker, 1986]. Shared memory refers to a multiprocessor system in which all processors share a common main memory. Such systems typically suffer from increased error-proneness as the failure isolation between processors is reduced according to [Rahm, 1993]. In shared disk architectures processors have sole access to their corresponding main memory but share a common disk. This reduces failure isolation of the processors but still introduces a common disk controller as single point of failure. According to [Stonebraker, 1986] only shared nothing architectures seem feasible in terms of failure tolerance, as in such systems data is usually partitioned and replicated amongst several completely independent database servers. Similar to single server DBS, such architecture typically lack the ability to exploit spatial locality.
- **NoSQL technologies:** Recent development efforts have brought forth several new systems that provide good performance and scalability for simple database operations distributed over multiple servers. [Cattell, 2011] provides an overview on the most popular representatives of these so called *NoSQL* systems. The diversity of existing systems ranges from simple in-memory key-value stores, such as Memcached which is used by Facebook [Nishtala et al., 2013], to more advanced technologies, such as Google's BigTable [Chang et al., 2008]. However, the process of scale-in and scale-out currently involves a lot of manual effort in such a system, as observed by [Konstantinou et al., 2011]. In general it can be observed that transparency towards application logic is often not

	Avail.	Spat. Sup.	Transp.	Locality Exploitation		
				intra	inter	spatial
Client cache	X	✓	✓	✓	X	X
Single DBS	X	✓	✓	✓	✓	X
Parallel DBS	✓	✓	✓	✓	✓	X
NoSQL	✓	(✓)	(✓)	✓	✓	X

Table 1.1: Support level of technologies for specific requirements.

the focus of NoSQL systems. Using Memcached, for example, the application must explicitly insert and remove data into the cache. In addition, the overall support of NoSQL systems for spatial processing is still low. Some of them, such as MongoDB, have started to include support for simple spatial operations [Plugge et al., 2010]. To our knowledge, none of the NoSQL solutions considers spatial locality.

Table 1.1 summarizes the support level of the technologies described above for specific requirements. The symbol "✓" denotes good support, "(✓)" limited support and "X" no support for a specific requirement. It can be observed that none of the mentioned technologies supports all requirements without restrictions. In particular, it has to be noted that Parallel DBS and NoSQL technologies seem to be the only ones that provide practical solutions to achieve high availability. An additional observation can be made with regard to locality exploitation. While some approaches exploit intra and inter client locality, none seems to pay special attention to spatial locality, a type of locality which is very likely in our application field. On spatial locality, we focus the attention of this thesis and propose a caching approach particularly suited for data spatial access patterns.

1.4 Contributions

In this work, we propose techniques towards a distributed spatial cache between the clients and the data back-end as shown by Figure 1.1. The distributed spatial cache comprises several cache servers which keep the most relevant data in their main memory using a dedicated cache strategy. In this

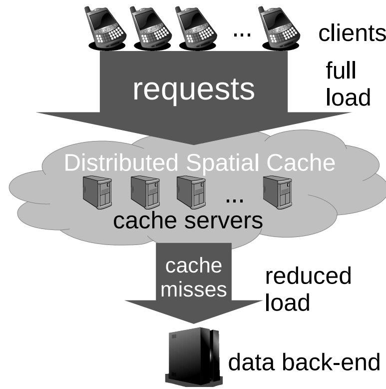


Figure 1.1: System overview

manner, it is the intention of such a strategy to select the most appropriate data for expected future data access patterns. Very often locality in the data access patterns can be exploited for this task. A successful caching strategy will thus significantly reduce the total load handled by the data back-end, as the data back-end only needs to be contacted when the data is not found in the distributed spatial cache.

To achieve the requirements defined in Section 1.2 (i. e. scalability, availability, elasticity, locality exploitation, transparency and spatial support), we applied several methods. In the following, we concretize the challenges in implementing the methods in the context of the just described caching architecture and outline our contributions in addressing these challenges:

- **Horizontal scaling:** First of all, through addition and removal of cache servers, resources can be added to or removed from the system: a process which is typically denoted as horizontal scale-in or scale-out respectively. The scaling process must have as little effects on the overall system as possible so that it can be performed at full operation of the system. This also includes the transfer of state between cache servers during scale-in and scale-out, as new servers with empty ("cold") caches must be filled up with relevant data for proper opera-

tion and leaving cache servers must hand over their cache content to the remaining servers so that future requesters of this data can still be served.

- **Vertical scaling:** In contrast to horizontal scaling, the process of increasing or decreasing the individual capabilities of single cache servers is usually denoted as *vertical scaling*. Prominently used in virtualization environments it allows adding or removing arbitrary resources (e.g., main memory or CPU cores) to virtualized processing nodes. Especially in regard to our approach this feature is required, as in the context of caching memory and processing capabilities have to be scaled separately. Scaling memory is required when, for instance, a poor cache hit-rate indicates that the overall cache capacity is too small to store all relevant data. Scaling processing capabilities is required when, for example, long processing latencies indicate an overload situation.
- **Flexible content allocation:** The question, on which cache server what portion of data should be stored, is intrinsically related to scaling. In principle two extremes are possible. First, each data item is stored by all servers (full replication) or second, each data item is stored on exactly one cache server (full partitioning). In reality, a trade-off between the two extremes is often practicable. The actual grade of replication and partition density has to be reconsidered during the scaling process. A typical question arising during scale-out is, for example, whether new cache servers should be devoted to store data which is already cached by other servers (thus increasing the replication rate) or should they be used to increase the overall cache capacity allowing to cache yet unconsidered data (thus increasing the partition numbers)?
- **Load balancing:** To achieve elasticity, load-balancing plays an essential role. The application field we focus on is characterized by high dynamics: With changing popularity of data over time, the data access patterns change and thus the frequencies of requests relating to certain spatial regions – a phenomenon, typically referred to as *workload skew*. Moreover, the information density of spatial regions is not

spread evenly throughout the globe, as typically more information about densely populated regions, such as large cities, is available than about unpopulated regions, such as deserts – a circumstance, usually denoted as *data skew*. A closer examination reveals the dependency of these two aspects in the context of load-balancing: Changing data access patterns may continuously mark spatial regions with varying data density as most relevant. Thus, the required memory to store all relevant data will depend on both the actual workload and the data skew at the same time. In the context of caching, a successful load-balancing algorithm must therefore consider both aspects.

- **Cache replacement:** Whenever the cache capacity is exhausted, data has to be cleared out of the cache in order to make space for new data – a process most commonly denoted as *cache replacement*. To make a proper replacement decision the locality in data access patterns plays an important role. For this, traditional algorithms often exploit temporal locality, i.e., the property that data which has been requested recently is likely to be requested again in the near future. As the data access pattern in our application field also possess spatial locality as pointed out in Section 1.1, our replacement algorithm uses a notion of spatial nearness on top of temporal locality to make the replacement decision.
- **Distributed processing:** To achieve availability and fault tolerance, a distributed system design is essential. Thus, the system state is distributed across a network of loosely coupled cache servers in our system. We devised a set of sophisticated protocols to establish and maintain this network. Thereby, special attention is focused on the issues related to distributed query processing: How should the network be organized to make efficient retrieval of data possible, in particular with regard to range queries? And for a given request, which are the most appropriate cache servers to answer the request and how can this cache server be reached within the network.
- **Layered software architecture:** To implement transparency, we devised a software architecture which groups closely coupled functionality in separate layers. Through well-defined interfaces between the

layers we achieve full inter layer transparency. Thus, neither scaling the distributed cache towards changing demand, nor the failure of a system component, nor the adaption towards changed data access patterns is noted by the clients sending the requests. A simple declarative query language allows the client applications to abstract from tasks which are specific to data provisioning and to focus on the application's original goal, i.e., to extract relevant data for the user.

- **Spatial index structure:** To support efficient spatial processing, sophisticated index structures are required. At the same time these index structures must allow to identify those parts which are currently not cached. This property is essential to construct a request to fetch all the missing data from the data back-end whenever a given query could not be fully served by the cache. In this work, we designed a simple grid-based spatial index structure supporting efficient processing of spatial window queries. In addition to that, the simplicity of the grid structure allows constructing computationally simple back-end requests which speeds up query processing at the data back-end.

1.5 Outline of this Thesis

The methods described in the previous section can be seen as a basic tool set to reach the overall objective, i.e. the requirements defined in Section 1.2. The entire task area can be divided into the parts that can be accomplished by a single cache server and the parts where cooperation between different cache servers is required. This basic distinction is reflected in the structure of this thesis, as outlined in the following.

Chapter 2 provides a system overview of our distributed spatial caching architecture. For this purpose, it explains the key fundamentals and assumptions of this work. Moreover, the chapter introduces various architectural approaches, discusses their main characteristics and assesses their suitability for this work.

In Chapter 3, we detail on the design and implementation of the local caching mechanism used on a single cache server. In the course of this

chapter, we discuss the general challenges in providing generic cache index structures and present an index structure optimized for processing spatial window queries. In addition, we assess different replacement strategies and present a method to identify the back-end parts for a given query.

Subsequently, we show how to combine multiple local server caches in order to form a distributed cluster of cache servers in Chapter 4. We present three different approaches and compare them through a qualitative and quantitative evaluation.

Chapter 5 briefly introduces prototypical implementations that emerged during this work. The prototypes vary from simple example applications from our target application field to sophisticated evaluation prototypes used for quantitative analysis of the systems performance.

We conclude this thesis in Chapter 6 and identify challenges and aspects for future research in the context of our work.

CHAPTER 2

Design Considerations for a Distributed Caching Architecture

Caching has always been an important ingredient in many hardware and software architectures. From a general perspective, a cache is a storage for transient data. In our context, the term "transient" indicates that the data is currently on its way from a data provider to a data consumer and is put at some intermediate place where the consumer can access it. This intermediate place, namely the cache, typically stores only a small selection of the data which the data provider offers. The deliberately introduced level of indirection can have advantages over direct access, because the cache has properties which the data provider can't offer on its own. Access speed, for instance, is a property exploited in the multiple cascaded levels of CPU caches of modern hardware architectures. Thereby, frequently used data is stored in high-speed memory components so that future requests for that

data can be served faster. Beyond access speed, additional merits can be achieved with the designated use of caches in distributed architectures. For instance, proxy caches in the World Wide Web increase the availability of highly frequented web pages and decrease the overall network contention. Many other examples for the effectiveness of caching exist.

In this work, we propose a cluster of dedicated cache servers forming a *distributed spatial cache* in between the data provider and its data consumers. Such a distributed spatial cache can be designed in various ways. Design decisions can have significant impact on performance relevant system properties. It is the purpose of this chapter to illuminate the key design dimensions of such a system, to discuss several possible implementation strategies and to assess their implications and suitability for our target application field.

In the following, Section 2.1 starts with an overview on the main system components. Then, Section 2.2 focuses on the design of a single cache server. Relevant design criteria for the distributed mechanisms necessary for operating a distributed spatial cache are discussed in Section 2.3. We propose a system architecture in Section 2.4. Finally, Section 2.5 summarizes the chapter and gives an outlook on the remainder of this work.

2.1 System Environment

Introducing a level of indirection in between the data provider and the data consumers allows the separation of concerns in the organizational structure of the system environment. Figure 2.1 shows the general intention. In this perspective, the data provisioning task is accomplished by two separate institutions. The data provider, on the one hand, is responsible for managing the data stock including typical tasks such as data acquisition, data maintenance, persistency, backup and recovery services etc. On the other hand, the cache provider is able to focus on ensuring scalability, availability and elasticity towards the application. Such organizational specialization may significantly increase the efficiency of the data provisioning process, as each organizational entity can focus on its key abilities. It is not our intention to

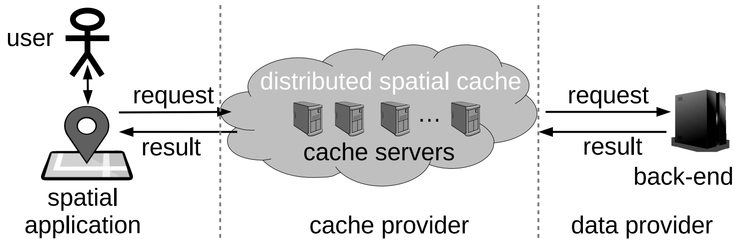


Figure 2.1: System Environment

discuss further economic advantages of such a organizational structure. We just argue that our system design facilitates such organization structures from a technical point of view.

Technically, the system environment comprises a back-end which stores the complete data stock. To transport the data from the back-end to the cache servers and eventually to the clients, certain data models and formats are used, as described in Section 2.1.1. To request data, a descriptive query language is used, as outlined by Section 2.1.2. Finally, Section 2.1.3 covers data maintenance aspects and their implications on the cache coherence.

2.1.1 Data Representation and Back-end Technologies

In this work, the focus is on caching spatial data. Spatial data typically describes real world objects in the geographic space, their relations and properties. Scientists have developed sophisticated models to encapsulate these aspects. An overview can be found e.g. in [Güting, 1994]. In principle many different options are reasonable to store and manage spatial data. Conceivable systems range from traditional relational [Stonebraker et al., 1976] over object-relational [Stonebraker and Moore, 1995] to object databases [Kim et al., 1989]. It is even possible to store data partially in separate, specialized subsystems. For instance, a key-value store [Cattell, 2011] could be used to manage the descriptive information (such as name or type information of a real world

object), a column store [Stonebraker et al., 2005] to store the numerical information (e.g. rating information) and a dedicated geographic information system to manage spatial information.

In each of these systems data is represented and accessed in an particular way. In this work, it is not our intention to commit to a particular technology. For this reason, we define an abstract data model which is inspired by the concepts presented in [Nicklas et al., 2004]. The data model is object-oriented, as this is an intiutive way to represent spatial data, i.e., real world objects. In this model, an *object instance* is a set of attribute instances. An *attribute instance* is a tuple consisting of an attribute name and a value. A dedicated attribute instance named, *extent* carries the spatial information in the WKT format [Open Geospatial Consortium Inc., 2011]. For instance, a shop named "IBU" with an average visitor rating of 90 % and a building outline delineated by a certain WKT polygon may be represented as follows:

$$\{\texttt{name} = \text{"}\texttt{IBU}\text{"}, \texttt{type} = \text{"}\texttt{Shop}\text{"}, \texttt{rating} = 90, \texttt{extent} = \text{"}\texttt{POLYGON}(...)\text{"}\}$$

Regardless the abstractions and data models a specific data store offers, it is generally possible to represent the information with the abstractions our abstract data model offers: objects and attributes. As a proof of concept, we converted spatial data from the relational data store of OpenStreetMap[1] into a format representing our object-oriented data model. Therefore, we are able to abstract from the specific characteristics and models of the differing systems. In this work, we assume that the information can be represented with our model regardless of the storage technology used in the back-end. The data from the data back-end can be requested via a query language as outlined in the next section.

2.1.2 Query Representation and Evaluation Strategy

Based on abstract data model presented in Section 2.1.1, queries are formulated as Boolean expressions over predicates using the Boolean operators $\wedge, \vee$, and $\neg$. A predicate is expressed as $A \phi C$, where A is an attribute name,

[1] http://www.openstreetmap.org, accessed on Apr. 17, 2017

C a constant and ϕ a comparison. Comparisons can be formulated over multiple data domains, such as numerical ($=, \neq, \leq, \geq$, etc.), string (*eq*, *neq*, etc.) or spatial (*within*, *intersects*, etc.). For instance, a query to retrieve all shops which have an average rating of more than 50% and which are located in a certain region – e.g., the region surrounding the user's current position given by the rectangle G – could be formulated as follows:

$$\texttt{type}\ eq\ \texttt{Shop} \wedge \texttt{rating} > 50 \wedge \texttt{extent}\ intersects\ G$$

As it can be observed, the query mentioned above includes various domains such as numerical, string and spatial. To process such compound queries several evaluation strategies can be applied. The task of finding an optimal execution plan for a given query is not trivial, especially when several index structures support the efficient and selective retrieval of data are available. For instance, it is debatable whether non-spatial attributes or the spatial attributes of the previous query should be processed first. In the first case, the query processor would use conventional indices to retrieve a collection of candidate objects and then check each found object against the spatial predicate. In the second case, the processor would use a dedicated spatial index to retrieve the candidate set.

The optimality of query execution plans is a complex research area, as an optimal execution order depends on several relevant factors, such as the selectivities of involved predicates, the available indexes and so on. We refer interested readers to [Chaudhuri, 1998, Chaudhuri, 2007] for an overview. The query optimization problem becomes even more difficult when data is distributed among several data stores. Therefore, we consider query optimization beyond the scope of this work. In the case of the aforementioned example query, however, the first plan obviously seems suboptimal, as the spatial predicate is intuitively more selective than the other two predicates, particularly when the region of the user's surroundings is small. For this reason, we limit our attention towards the second strategy and always consider the spatial domain first. As our target applications typically use location as their primary selection criterion in the field of LBS, we expect this is an adequate heuristic.

2.1.3 Data Maintenance

In our application field, data providers are confronted with a read-intensive workload of spatial window queries. To handle the read requests, we introduce a distributed spatial cache in between the clients issuing the requests and the data back-end which stores all data. In the following, we discuss issues concerning data updates. Typically updates occur in two situations: First, to maintain a constant data quality, data providers are obliged to run a continuous data cleaning process as integral part of the data management. In this usually at least partly automated process, data quality and error models are used to detect and correct data of poor quality [Hunter et al., 2009]. And second, data providers very often actively incorporate their customers feedback to improve their data stock. Quite typical is the outsourcing of quality assurance tasks in which users assess data quality and even suggest changes via a on-line recension system. For instance, Google Maps[2] is pursuing this approach. Others, such as OpenStreetMap[3], go a step further by treating the users as the actual data producers. In this community-managed data stock, users can include own data, e.g., the GPS trace of the last bike trip. Whether updates are triggered by an automatic cleaning process or by the community of users, in both cases an asynchronous update process typically applies accumulated update requests in a batch-wise manner to the master data set.

With the updates comes the problem of data consistency: Once data is copied into the cache, it runs the risk of becoming stale as soon as the original data is updated. To counteract this tendency specific protocols can be used with the objective to maintain consistency between cached and original data. In other research fields, e.g., the field of web-caching, the cache consistency problem has been exhaustively studied. In principle, two variants are possible in our distributed cache architecture: First, with a *cache-driven* protocol, the cache servers would periodically send validation requests to the data back-end to find out whether the cached data is still valid [Krishnamurthy and Wills, 1997]. And second, with a *back-end-driven*

[2] http://maps.google.com, accessed on Apr. 17th 2017
[3] http://www.openstreetmap.org, accessed on Apr. 17th 2017

protocol, the data back-end would have to send invalidation reports to the cache servers to inform them about stale data [Yin et al., 1999].

Depending on the amount of additional effort, the protocols can guarantee different degrees of consistency. Strong consistency guarantees [Liu and Cao, 1998], i.e. the cached and original data is consistent all the time, are typically the most expensive ones in terms of message overhead. Weaker consistency notions, such as the one defined by [Urgaonkar et al., 2001], guarantee that cached and original data is consistent according to some consistency bound, e.g., the cached data is not more than two hours old. Such consistency guarantees are certainly cheaper in terms of additional overhead. In accordance with the arguments presented in the characterization of our application field (see Section 1.1), we do not expect many location-based applications to have strong consistency requirements. For this reason, cache consistency is not the focus of this work. However, if consistency guarantees are required in the future, well-known consistency protocols from the field of web-caching can be adapted for this purpose.

2.2 Designing a Cache Server

In this section, we discuss the main issues relevant for the internal design of a single cache server. Its main purpose is to cache extracts of the data stored at the data back-end. In the following, Section 2.2.1 details on the cache server's mode of operation. The internal cache management can operate on several different data granularities which may have a serious impact on the cache server's performance. Section 2.2.2 examines different approaches and assesses their suitability in the context of our application field. Section 2.2.3 illuminates several approaches for data search in the cache server's internal cache. Section 2.2.4 outlines several caching strategies which determine what content should be loaded into the cache. Finally, we discuss in Section 2.2.5 which memory type is most suitable for our purposes.

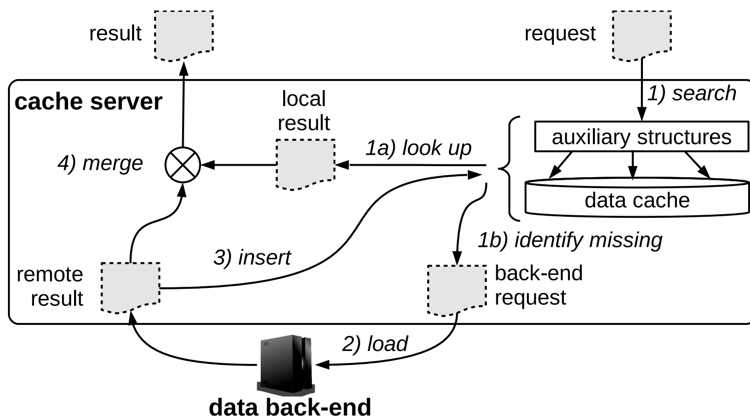


Figure 2.2: A cache server's internal design

2.2.1 Operation Mode

This section details the operation mode of a single cache server. Figure 2.2 provides an overview. The main component of a cache server is its *data cache*. It resembles a chunk of memory where it can store cached data. To efficiently access this data, a cache server may use additional data structures, denoted as *auxiliary structures* in Figure 2.2. A cache server has to perform the following actions in order to process a request and to return the corresponding result:

1. **Cache search:** To process a request, the cache server has to search the data cache. During the cache search, two tasks have to be accomplished:

   a) The cache server looks up all data qualifying as a result in its data cache. Subsequently, this data is copied to a temporary data structure, denoted as *local result* in our Figure 2.2.

   b) All requested data which could not be found in the data cache, has to be identified. For this data, the cache server then generates a back-end request.

2. **Loading of missing data:** The previously identified missing data is loaded into a data structure containing the remote result of the back-end.
3. **Inserting data:** The remote result is inserted into the data cache. When the maximum capacity of the cache is reached, the cache management must overwrite existing data, to insert the retrieved back-end data. For this purpose it uses a dedicated caching strategy.
4. **Result preparation:** The data retrieved from the local cache (*local result*) as well as the data from the back-end (*remote result*) are merged. After optional final adjustments, such as sorting or duplicate elimination, the result is given back to the client.

To carry out the previously described tasks, the granularity of cached data entities is of central concern for the cache management. We discuss several design variants in the following section.

2.2.2 Data Granularity for the Cache Management

For caching architectures the data granularity of the cache management is a major design issue. Existing approaches have brought forth several alternative granularity levels. For instance, tuple caching approaches operate on single tuples while page caching use whole pages as basic unit for caching. We compare the different approaches according to the following performance-relevant criteria which are affected by the granularity level:

- **Identification Accuracy:** This criterion determines the accuracy for identifying data chunks for replacement or for the data loading process. The data granularity of cache management influences this accuracy. Generally speaking, caching at a fine-grained level increases the data identification accuracy while a coarse data granularity decreases it. High data identification accuracy increases the ability to react to specific data access patterns during replacement, as data can be replaced in a more accurate manner. Moreover, it facilitates exact retrieval of data chunks from the data back-end during load. Inaccu-

rate loading leads to decreased replacement flexibility and increased data transfer from the back-end to the cache.

- **Management overhead:** Typically, the cache granularity determines the space overhead needed for cache management (free memory lists, translation tables, hash tables etc.). A coarse granularity simplifies the cache management and reduces the space overhead.
- **Loading effort:** The data granularity also affects the complexity of back-end requests. This influences the effort needed for loading data from the data back-end. For one, the cache server has to spend some effort in creating a back-end request for loading missing data. For another, the back-end has to process this request. The more complex the back-end request is structured, the more effort is induced during the load process.

In the following, we examine the properties of several caching architectures in respect to the above mentioned criteria. Subsequently, we compare the different approaches and discuss their suitability in the context of our work.

2.2.2.1 Tuple / Object

In tuple caching architectures, such as described by [DeWitt et al., 1990], the data cache is organized as a collection of single tuples or objects respectively. Caching at this fine granular level enables accurate data identification. This allows very specific coordination of replacement and thus allows to adjust to individual data access patterns. Moreover, it enables exact retrieval of missing data during load. To load missing data for processing a given client request, a list of all cached tuples which qualify as a result is sent to the data back-end along with the request constraint. With this information the back-end can return an exact partial result containing only the result tuples which are missing at the cache server. However, the tuple list can become very large when a lot of cached tuples qualify as a result. As this causes very large back-end requests, this significantly increases the loading effort for retrieving partial results from the back-end. Moreover, the space overhead needed for cache management (free memory lists, transla-

tion tables etc.) is proportional to the total number of tuples fitting into the cache which is usually large.

2.2.2.2 Page

In page caching architectures, such as that proposed in [Carey et al., 1994], the tuples are grouped into pages of fixed size. This significantly reduces the cache management overhead, as the total number of managed pages is much smaller than that of tuples. However, it reduces the data identification accuracy, as data can only be allocated in terms of pages, i.e., in chunks of multiple tuples. In general, the tuples within a page are not semantically clustered so that similar content can be spread across multiple pages. Thus, retrieving data from the back-end in units of complete pages can lead to unnecessary data transfers: In the worst case, a single qualifying tuple may cause a complete page transfer containing thousands of tuples which do not qualify as a result. To load missing data, a list of all required pages is included in the back-end request. If the required data is distributed among several pages, this page list can become very large. This increases the overall effort during load.

2.2.2.3 Request

Modern database systems, such as MySQL, use a dedicated result cache on top of the traditional database buffer, as described by [Pröll et al., 2013]. Such a cache is organized in key-value fashion where requests are interpreted as keys and the corresponding results as values. The loading effort of this approach is small, as a request is simply forwarded to the back-end, if it is not found in the cache. Moreover, with a hash table based implementation constant cache lookup times can be achieved. Also, cache management overhead is within acceptable limits, as the number of key-value entries is proportional to the number of cached queries and thus typically smaller than that of the tuple and page caching approaches. The data identification abilities of request caching approaches are very limited, as a cached result can only be re-used if the corresponding cache entry matches a client

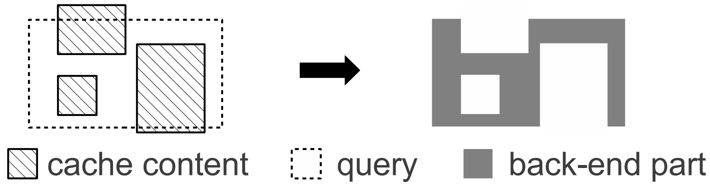


Figure 2.3: Complex back-end part of a spatial query

request bit for bit. For location-based applications, this is quite unlikely, as the location context is usually derived from sensors whose values vary greatly. As the replacement is carried out in units of complete requests, the approach lacks the flexibility to adapt to the predominant data access patterns in our target application field.

2.2.2.4 Predicate

Some caching architectures use predicate expressions to group relating tuples into logical units. Representative examples of these architectures can be found in [Dar et al., 1996] or in [Härder and Bühmann, 2008]. The general idea behind this approach is that the cache can be managed using predicate expressions. Thus, given the predicate expression Q_i of a query and the cache content C_{i-1} that was valid before processing the query, one can derive predicate expressions for the cache and back-end part. The *cache part (CP)* includes all cached objects satisfying the query: $CP(Q_i) = Q_i \wedge C_{i-1}$, and the *back-end part (BP)* comprises all objects satisfying the query which are not currently cached: $BP(Q_i) = Q_i \wedge \neg C_{i-1}$. With these prerequisites a predicate expression describing the current cache content can be formulated. Let $C_0 = false$ express the empty cache, let $Q_1, ...Q_k$ denote queries which are sequentially inserted into the cache and let the cache have sufficient capacity to fit in the k query results. Then the cache content C_k after processing the k-th query can be expressed as a disjunction of all present back-end parts: $C_k = BP(Q_1) \vee BP(Q_2) \vee \cdots \vee BP(Q_k)$.

The predicate expressions allow exact identification of object sets. Thus, this approach achieves high cache utilization in particular for partial hits. Moreover, the theoretical flexibility during replacement is very high, as basically any objects can be selected for replacement as long as the cache content expression is modified accordingly. However, the complexity of the cache content expression C_k increases over-proportionally with the number of processed queries in the worst case, as the number of involved terms can be approximated by the triangular number $n = 1 + 2 + ... + k = \frac{k(k+1)}{2}$. Therefore the overhead for cache management is high with this approach, especially when a lot of queries have to be processed. Moreover, the spatial dimension is difficult to handle with this approach, as continuous insertion of spatial predicate expressions creates complex geometric constellations of the cache content. This situation becomes even worse when the cache needs to replace parts of the cache content in order to free space for new queries. Along with the increase in complexity of the cache content, the complexity of the predicate expression of the back-end part will increase significantly, as indicated in Figure 2.3. This in turn increases the loading effort for both the data back-end and the cache server.

2.2.2.5 Partition

Another possibility is to group related data into partitions using logical criteria. In contrast to predicates whose expressive power is very high, the criteria for creating partitions are much simpler. A common practice is to partition objects in respect to certain value ranges of a scalar attribute dimension or to group them according to a parcel-wise partition layout of geographic space.

In a partition-based caching architecture, data identification accuracy strongly depends on the partition size. In contrast to pages, partitions are clustered to a certain semantic criterion. The semantic grouping of data within partitions is advantageous, as whole semantic regions can be effectively identified. Thus, the effectiveness of data identification during replacement and for loading is increased. Comparable to page caching, the overhead for cache management stays within reasonable limits, as it is pro-

Granularity	Id. Accuracy	Mngt. Overhead	Loading Effort
Tuple/Object	++	-	-
Page	-	+	-
Request	-	++	+
Predicate	++	-	-
Partition	+	+	+

Table 2.1: Comparison of cache granularities.

portional to the number of partitions that can be cached. In contrast to the predicate approach, however, the overhead is not dependent on the processed queries which is advantageous in our context. The simplicity of the semantic criteria used for partitioning is also advantageous in the context of loading, as it enables formulating back-end requests in a very concise manner. For instance, if objects are partitioned according to the scalar attribute price into the value ranges of 5 € per partition, then the first 12 partitions can be delineated by the simple predicate `price` ≤ 60. Using such synoptic predicate expressions, the effort for cache loading compared to the page caching approach can be significantly reduced.

2.2.2.6 Assessment of the Granularity Variants

Table 2.1 summarizes our findings. We graded the different approaches with respect to the examined criteria (identification accuracy, management overhead and loading effort), where ++ denotes excellent, + fair and - bad performance. As it can be observed, the granularities tuple, page, request and predicate induce severe disadvantages at one or more of the examined criteria in the context of our target application field. The partition approach offers fair performance for all examined criteria. By adjusting the partition sizes, the performance of single criteria can be adapted towards specific requirements. For this reason, we consider partitions as the most appropriate granularity level for caching in our application field. Our contribution introduced in Chapter 3 constitutes a partitioning approach for geographic data.

2.2.3 Search Mechanisms

Remember that the cache server has to search its data cache to process a client request. During this process all cached data qualifying as a result is looked up and potentially missing data is identified. The following sections detail on these issues.

2.2.3.1 Look Up of Qualifying Data

[Effelsberg and Haerder, 1984] classify the possible algorithms into *direct* and *indirect* look up mechanisms. The first (direct), performs a sequential search in the whole data cache. The second (indirect) uses additional data structures to speed up this process. In traditional database buffer management systems, these auxiliary structures range from simple translation or hash tables to balanced trees serving as indices.

For the spatial domain, specialized index structures have been invented; the most prominent is the R-tree proposed by [Guttman, 1984]. The R-tree is a depth-balanced tree. To maintain the balance of tree nodes, costly split and merge operations have to be carried out when data is inserted or deleted from the tree. In the caching context, these operations are quite likely, as the cache replacement continuously adapts the cache content towards current access patterns. Against this background, we use a hash table-based index structure in our solution presented in Chapter 3, as hash tables perform well under frequent inserts and deletes.

In the context of our application field, the evaluation of spatial predicates is computationally intensive, when the required tasks (e.g. polygon intersection, containment tests and others) involve complex geometric algorithms. For this reason, the filter and refine paradigm is often applied to evaluate spatial queries [Park et al., 1999, Wood, 2008]. Thereby, a preliminary filter step removes data which cannot possibly contribute to the result. On the thus obtained small subset of the original data, the computationally intensive tasks are executed subsequently. In our contribution presented in Chapter 3, we take advantage of this principle and deliberately design

the cache index structures in a way so that a fast pre-selection of data is possible.

2.2.3.2 Identification of Missing Data

There are three main approaches to identify those data parts referenced by a request which have to be loaded from the data back-end:

- **Fault list approach:** In the fault list the cache server includes all data entities which are missing to answer a given request. For instance, the caching architectures described in [DeWitt et al., 1990] use this approach.
- **Hit list approach:** The theoretical counterpart of the fault list approach is the hit list approach. In this approach, the cache server sends a list of all data entities which qualified as a result to the back-end along with the complete request predicate. With this information, the back-end can identify those data portions which are still needed at the cache server to answer a request.
- **Predicate derivation approach:** In this approach, the cache server derives a predicate from the current cache content and the client request that logically describes the missing data parts. The semantic cache architecture described in [Dar et al., 1996] follows this approach.

The hit list approach, as well as the fault list approach requires that a global unique identifier is maintained between cache server and back-end. In addition, the cache server needs to be aware of all data stored at the data back-end in the fault list approach. In general, these requirements introduce inflexibility in the cache management. Moreover, the hit list as well as the fault list can become quite space consuming when large amounts of identifiers have to be included into the back-end request. With predicates this effort is reduced, as even simple predicates can identify large amounts of data. As we consider this property as useful in our context, we follow the predicate derivation approach in our contribution presented in Chapter 3.

2.2.4 Caching Strategies

To make caches useful, they should always store those data portions which are most likely expected to be referenced again in future requests. To achieve this goal, different caching strategies exist. These can be classified into *proactive* and *reactive* strategies. In the following, we briefly sketch the main characteristics of these two classes and discuss their suitability in the context of this work.

2.2.4.1 Proactive Caching

In proactive caching strategies [Hu et al., 2005], also sometimes referred as the synonyms prefetching [Smith, 1978] or hoarding [Kubach and Rothermel, 2001], the cache is actively prepared for expected data access patterns before the actual request processing takes place. Typically, this strategy is applied in client-side caches where future access behavior can often be inferred from context information, such as the user's current location. Then the client-side cache can be filled with relevant data before the user issues the actual request. For server-side caching, prediction of future access patterns is more difficult, as the behavioral patterns of all users have to be incorporated. To find an adequate prediction model in a distributed caching architecture is even more complex, because such a model would not only have to predict what data is requested next, but also where (on which cache server) this data should be stored. As an inadequate prediction would cause unnecessary data transfers between back-end and cache servers, we consider a reactive caching strategy as more appropriate in our context.

2.2.4.2 Reactive Caching

A reactive caching strategy loads data into the cache only when it has been requested. Thus, no information based on anticipated access patterns is transferred. In such a strategy, specific replacement algorithm controls the cache content. Once the cache is full and space for new data needs to be allocated, a candidate for replacement is chosen by the cache replacement

algorithm. In this process different selection criteria can be used. As also described in [Effelsberg and Haerder, 1984] common replacement algorithms for traditional databases include the age or the reference frequency of data into their selection decision. For instance, the first-in-first-out (FIFO) algorithm first replaces the data which has stayed in the cache for the longest time, and the least-frequently-used (LFU) algorithm prefers the data with the lowest reference frequency for replacement. In addition to the traditional criteria (age and reference frequency) [Dar et al., 1996] introduce semantic relation as replacement criterion. Semantic relation is thereby expressed by a distance function providing a measure of similarity between data entities. Thus, the replacement algorithm first replaces the data which is semantically least related to the most recent query. In this way, the algorithm exploits the fact, that users often send multiple similar requests in a row, for example, to successively refine their results. In the spatial domain, we consider such access patterns as very likely especially for mobile users that are bound to the physical constraints of traveling in space and time. However, when data is spread among multiple cache servers, the simple replacement algorithms mentioned above are not sufficient to organize the global replacement for all cache servers.

In this work, we introduce a new replacement concept which we denote as *focused caching*. In focused caching, a cache server replaces data with the highest Euclidean distance to a certain point in space denoted as its *cache focus*. With clever positioning of the cache foci of participating cache servers, the replacement can be organized globally. Chapter 3 details on this mechanism.

2.2.5 Cache Memory Types

Modern hardware architectures offer several types of memory. The most prominent representatives are random access memory (RAM), solid state drives (SSD) and conventional hard disk drives (HDD). They mainly differ in terms of random access behavior, speed, cost and persistency support. Their properties have been thoroughly examined by existing work, such as [Hudlet and Schall, 2011, Grochowski and Hoyt, 1996,

Technology	Random Access	Speed	Cost	Persistency
RAM	++	++	-	volatile
SSD	+	+	+	permanent
HDD	-	-	++	permanent

Table 2.2: Properties of Cache Memory Types.

Kiyoo Itoh et al., 1995]. From these works, we deduced the main characteristics of the different memory types as summarized in Table 2.2. In the following, we assess the suitability of these memory types for our purposes.

The first characteristic grades the support of a specific memory type for random data access patterns. Hard disks are good for sequential access and perform poorly for random access patterns. To perform well, data needs to be physically clustered on the hard disk in the same way as it is accessed. Maintaining the clustering within the data cache would be an intolerable expense, because we expect access patterns to change over time and because constant replacements will lead to a high fragmentation of the data clusters. Therefore, we consider hard disks as inefficient for our purposes.

At a first glance, SSDs seem to be a good compromise, as they offer a fair performance-price ratio. Their ability to store data permanently, however, is not a vital advantage in our context, as lost data can be re-loaded from the persistent data back-end anytime. Moreover, permanently cached data runs the risk of being outdated anyway after long server downtimes. As [Hudlet and Schall, 2011] observed in their measurements, some SSD devices still suffer from random access patterns even though this effect is not as tremendous as for HDDs. Also problematic in our context are the characteristics of SSDs under frequent write operations, as write operations are not unlikely in our context due to cache replacement. The endurance problems of former SSD devices caused by frequent write operations seems to have been mitigated by advanced wear leveling mechanisms today, as observed by [Boboila and Desnoyers, 2010]. However, writing on a SSD is still slower than reading in most cases. In conclusion, we regard SSDs suitable for our caching architecture only under the above mentioned limitations.

As of today, we consider RAM as the ideal memory type for implementing our caching architecture, as it offers superior random access performance. The higher costs for this type of memory stay within affordable limits for the data volumes required in our context. For instance, to store a complete and uncompressed extract of OpenStreetMap in memory would require about 740 GB of RAM[4]. Taking into account current hardware prices the necessary investment seems to be feasible even for small companies. Thus, our contributions are optimized for RAM. However, it is possible to transfer our concepts to SSD technology. This can be particularly reasonable for use cases demanding large-scale and fair-priced cache storage.

2.3 Designing a Cluster of Multiple Cache Servers

The previous section focused on the internal design of a single cache server. Multiple of such cache servers can be combined to form a distributed cache. In this section, we examine the design variants relevant for cluster-driven distributed caching architectures.

For this purpose, we describe typical operational tasks of such a distributed system in Section 2.3.1. Afterwards, we classify different design approaches according to the degree of autonomy of involved system components in Section 2.3.2. Subsequently, Section 2.3.3 discusses several approaches for clients to access computer cluster executing the distributed cache. Finally, we focus on aspects relevant for the internal network design of the cluster in Section 2.3.4.

2.3.1 Main Tasks during Operation

During the operation of the distributed spatial cache the following tasks have to be accomplished:

- **Cluster Maintenance:** In a distributed system, failures of single components happen regularly. Therefore, the whole system has to be

[4] cf. http://wiki.openstreetmap.org/wiki/Planet.osm, report date 29th, Nov 2016

constantly monitored to detect possible malfunctions of system components. If a component fails, it has to be replaced or removed from the system.

- **Scaling:** To adapt to changing loads, resources (e.g. memory, computation capability) have to be added to or removed from the system. The main challenge is to decide the time and the extent of the scaling action. Typically, global metrics measuring the current load and the overall utilization of involved resources have to be acquired to make a proper scaling decision.
- **Load-Balancing:** A further possibility for handling dynamic load is load-balancing. Its goal is to balance the load among the system components as evenly as possible. The main challenge during this task is to identify and eliminate load imbalances among the system parts. Typically, load-balancing decisions base on metrics characterizing the current utilization of the components.
- **Content Arrangement:** The content arrangement of a distributed data store determines which server is responsible for which data. In combination with the user's data access behavior, it effectively influences the workload of single system components.
- **Query Processing:** In our system, query processing includes retrieving the requested data from possibly different locations of the distributed cache and from the data back-end if required. To identify the locations where to search for relevant data is a key challenge in this task.

Most of these tasks are interrelated and have to be accomplished in a distributed manner. In the following, we discern different grades of distribution for such tasks in a distributed system.

2.3.2 Degree of Distribution

A distributed system can be designed in several ways. The main distinguishing criterion is typically the degree of distribution in the system design,

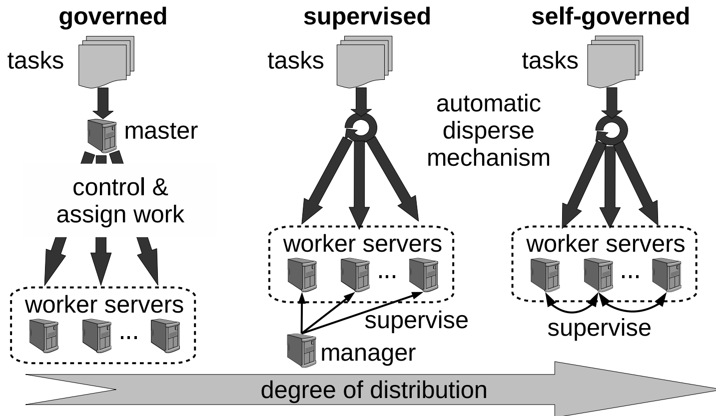


Figure 2.4: Design variants for the distributed spatial cache

ranging from full centralization to massive distribution. Figure 2.4 shows three conceivable design approaches.

In the *governed* approach, a central master component governs subordinate worker servers which execute most of the tasks with direct involvement of the master. Google's Map-Reduce approach as described by [Dean and Ghemawat, 2008] is an example for this design variant. In such a design, the master is typically responsible for distributing and balancing the workload, monitoring the current system state, adding and removing new nodes and more. The direct involvement of the master component introduces a single point of failure in the core functionality of the system. Thus, the system immediately stops functioning when the master component fails. For this reason, additional effort has to be taken to ensure availability in case of a failure. Therefore, typically at least one identical back-up component continuously runs in the back-ground ready to overtake the master's function any time. To make this possible, the master's internal state has to be continuously mirrored which creates additional overhead.

In the *self-governed* approach, the cluster servers autonomously organize themselves and no central administration component is

needed. Sophisticated algorithms and protocols for autonomous management of distributed content stores have been proposed in the field of peer-to-peer processing. The most well-known examples can be found in [Stoica et al., 2001, Rowstron and Druschel, 2001, Zhao et al., 2001, Ratnasamy et al., 2001]. In such a system, the participating servers typically execute the operational tasks in a cooperative manner in which every server is equally important. Thus, each server is principally capable to serve as entry point for client requests which increases the flexibility of the cluster design. However, it is usually complex to obtain an aggregated overall system state in such a system, as the state is dispersed among the independent servers. Thus, tasks requiring such an aggregated view (e.g. workload-aware system scaling) are generally difficult to realize and computationally expensive.

The *supervised* approach constitutes a trade-off between the two extremes previously described. In this approach, a lightweight manager component supervises comparatively independent worker servers which are responsible for the lion's share of operational tasks. During a failure of the manager most parts of the system can still pursue their work which increases the availability of the system.

In our application field, we expect many clients issuing a great number of requests which have to be handled by the distributed spatial cache. Even in case of failures of single system components, the overall system must continue its operation. Guaranteeing high availability and throughput in the governed approach is generally hard to achieve, as the central master component constitutes a performance bottleneck as well as a single point of failure. To manage the distributed caching cluster, the master component has to keep a lot of dynamic internal state, such as the cache content of different cache servers, their current workload and so on. In consequence, a straightforward replication of the master component to ensure availability is not applicable. For these reasons, we consider the governed approach as impractical in our context. In this work, we aim for a system design which includes as little centralization as possible. Thus, we propose a self-governed approach for most of the operational tasks which are listed in Section 2.3.1. Only our proposal for the scaling operation includes a central

component for measuring the overall utilization of the system and making the scaling decision. However, the required internal state of the central component is kept at a minimum so that the availability can be ensured through replication of the component.

2.3.3 Access Mechanisms for the Cluster

The previous section discerned the optimal grade of distribution for our cluster-driven caching architecture. This section examines ways of how clients should access the cluster. The intermediate technologies mediating between the clients outside the cluster and the servers inside the cluster is often referred as the *cluster edge*. In the following, we highlight the pros and cons of several possible cluster edge designs.

- **Client-side Access Point Information:** In this approach, the client-software is shipped with access information which helps the client to find a suitable access point for the service. A very simple possibility is to provide list or range of IP addresses of cluster servers from which the client can choose one to access the service. Through client-side random address probing, the workload is distributed evenly among the provided user access points [Dinger and Waldhorst, 2009]. However, it must be ensured that the public IP addresses are always reachable. This introduces inflexibility, as these IP addresses cannot be used for other purposes. Moreover, the security in respect to malicious attacks on the exposed cluster servers must be ensured through sophisticated mechanisms running on each cluster server. This affects the performance and makes the cluster-side administration process complex. Because of the poor usability and security aspects, we consider this approach as inappropriate for a sustained cluster operation.
- **DNS:** The domain name system (DNS) is a distributed naming service which serves as a well-established standard in the Internet for translating textual domain names to numerical IP addresses. For reasons of availability and response time, the DNS system is highly redundant which causes DNS entries to be replicated at multiple places in the Internet. The available DNS infrastructure can be exploited

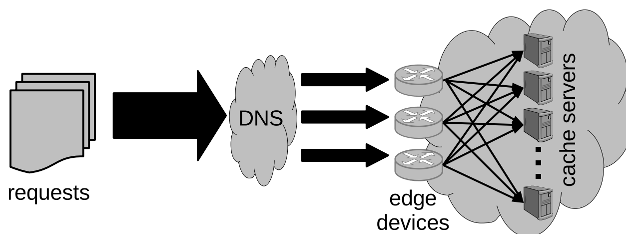


Figure 2.5: Proposed design of the cluster edge.

for providing access point information towards clients, as explained by [Brisco, 1995]. The idea is that a DNS server responds to a DNS request with a list of IP addresses of currently available cluster servers. The client can then pick one of these addresses to access the service. To keep the access point information up-to-date, the cluster manager can update the corresponding DNS entry. However, it takes time until the updated access information is eventually propagated towards the clients, as all replicas of the DNS entry have to be updated as well. Therefore, immediate reactiveness is not possible with this approach.

- **Edge Device:** Another common method is the use of dedicated edge devices serving as entry points for accessing the service. The edge devices forward incoming client requests to the internal cluster servers. In this design, the interior of the cluster is hidden from the clients, as only the access information about the edge devices is exposed to the clients. In addition to the gained flexibility in cluster management, this approach has serious advantages concerning security threats, as strict security requirements only have to be enforced on the exposed edge devices. However, the edge devices must be available all the time and able to handle high throughputs, as the total workload arrives at these entry points. Typically, this can be achieved through a redundant use of specialized edge devices supporting hardware-enabled forwarding mechanisms, such as round-robin.

For the purpose of this work, we propose a combination of DNS and edge devices, as depicted in Figure 2.5. In this approach, the service provider

publishes one domain name for accessing the service. The corresponding DNS entry contains as list of IP addresses of dedicated edge devices. Thus, in case of a edge device failure, the client can use another edge device as access point. The edge devices forward the incoming requests to the internal cluster servers. As only a single domain name is exposed to the clients, the flexibility for cluster management is maximized with this approach. Moreover, the two-level hierarchy of DNS and edge devices enables handling high request volumes. Finally, the architecture allows efficient application security standards in the edge devices which serve as gatekeepers to sebcure the whole cluster.

However, commercially available edge devices, typically only support IP-based forwarding mechanisms. As the clients of our application field typically request data with geographical content, an additional content-based forwarding mechanism is required to forward an incoming request to the appropriate cache server storing the content. Moreover, the distribution of load between the cache servers achieved through the simple IP-based forwarding mechanisms of the edge devices is not sufficient in our context. As a request for data eventually ends up at the server storing the content, the load balancing decision should be based on what content was requested instead of IP addresses. In our contributions presented in Chapter 4, we propose mechanisms for content-based routing and load balancing.

2.3.4 Internal Network Design of the Cluster

In the computer cluster executing the distributed spatial cache, effective communication between the cache servers is essential. This section outlines aspects of network design enabling such effective communication in a computing cluster. We examine the properties of certain physical network designs in Section 2.3.4.1. In Section 2.3.4.2, we discuss the consequences for logical communication structures which are built on top of the physical networks.

2.3.4.1 Physical Network Structure

The best known and largest example for a physical network structure is the Internet. The communication infrastructure in the Internet bridges very large distances. Thus, the physical limits of signal propagation time induce non negligible latencies for some communication relationships. In addition, high network traffic increases the packet switching time and therefore the perceived latency for some communication links in the Internet significantly.

Within a computing cluster, the signal propagation time is not an issue, as the communication infrastructure only needs to bridge short distances. However, network traffic is an aspect which requires intensive consideration in computing clusters, as the communication volume in between cluster servers is typically very high. For this reason, dedicated network topologies, such as the Fat-tree [Leiserson, 1985] or alternative network designs basing on a 3D torus [Costa et al., 2013] evolved. The basic idea of such network designs is to include massively redundant communication links, so that data can flow over alternative paths when single links are congested. With this method, very low latencies can be achieved, even in times of high network traffic.

Beyond hard-wired network designs, software defined networks (SDN) allow flexible provisioning of network infrastructure [Kim and Feamster, 2013]. With such a technology, the network infrastructure can be re-dimensioned to adjust to the currently demanded throughput.

In conclusion, the previously mentioned technologies enable low latency, high bandwidth point-to-point communication which is independent from the concrete network position of the communication partners and the current network traffic. These properties are important prerequisites to be able to abstract from concrete physical network designs and built logical communication structures on top of the physical network. In logical communication structures reorganization of communication links is much easier that in a hard-wired physical networks. We focus on such logical communication structures in the following section.

2.3.4.2 Logical Communication Structure

On top of physical networks allowing point to point communication, logical link structures, also denoted as overlays, can be established. In such a logical overlay, we say two servers are connected via a logical link when they mutually know their network addresses. Thus, the network topology is reflected by the connected components internal state, rather than by the wiring of network cables. Logical overlays are used in so-called content delivery networks [Li, 2008]. The most prominent implementations have been published in [Ratnasamy et al., 2001, Rowstron and Druschel, 2001, Stoica et al., 2001, Zhao et al., 2001]. In such a network, the content is distributed amongst multiple nodes. Clients can search for specific content using distinct keys. Typically, the network autonomously determines the corresponding node to answer a client request, so that content is delivered in a transparent manner regardless of the actual storage location.

Beyond the traditional content delivery networks which operate in a key-value fashion, extended query mechanisms are required in the field of LBSs. Very often clients search for data within certain geographic regions, instead of single keys. In the course of this work, we present the necessary mechanisms enabling geographic content delivery.

2.4 Proposed Software Architecture

The previous sections discuss possible design variants for the distributed spatial cache in respect to the system environment (cf. Section 2.1), the internal design of a cache server (cf. Section 2.2) and the cluster of multiple cache servers (cf. Section 2.3). In this section, we propose a general software architecture for the complete system landscape.

The software required to run the overall system composes a layered architecture, as depicted in Figure 2.6. Each layer provides a well defined interface to its upper layer and groups closely related functions.

The *application* runs on a client device, such as a smart-phone, laptop or possibly a desktop PC. The application constructs requests using the descrip-

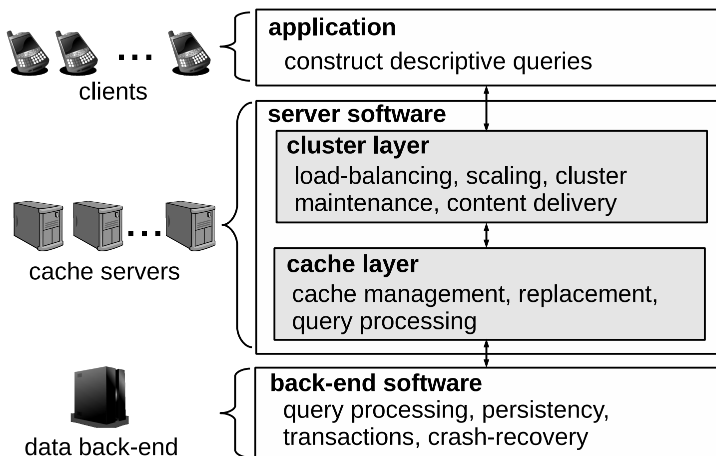


Figure 2.6: Proposed software architecture

tive query language defined in Section 2.1.2 and sends them to the cluster running the distributed spatial cache.

The *cluster layer* organizes the cluster of cache servers. It is needed for initial cluster construction, dynamic resizing of the cluster (scale-in, scale-out), load-balancing and the necessary cluster restructuring in case of cache server failures. It receives client requests, identifies appropriate servers for processing and delivers retrieved content back to the client.

The *cache layer* manages the cache server's local caches which basically store queries and their results in volatile memory. It also implements a specific replacement strategy which determines the cached data to replace when the cache capacity has been reached. After the cluster layer has identified the appropriate servers, a request is passed on to this layer for processing. The cache layer first processes a given request on cached data. If the request cannot be answered completely, it constructs a request to fetch missing data from the data back-end. It merges cached results with back-end results and returns the combined result to the cluster layer.

The *back-end software* is responsible for managing the complete data stock. Typical task fields include query processing, transactional operation, measures for providing persistency guarantees and crash recovery.

2.5 Summary and Outlook

In this work, we introduce a distributed spatial cache in between the data provider and its data consumers. This chapter discusses the suitability of several possible design variants of such a distributed spatial cache and thereby discusses existing work. For this purpose, the chapter inspects the system environment, elaborates the internal design of cache servers and discusses general aspects of the cluster design. Finally, a general software architecture for a distributed cache is proposed.

The proposed software architecture includes two essential layers: the *cache layer* and the *cluster layer*. As these layers encapsulate our contributions to the core challenges of our work, this thesis continues with a detailed description of them. Thereby, Chapter 3 proposes an implementation for the cache layer specifically suited for spatial data. Subsequently, Chapter 4 presents three possible implementation approaches for the cluster layer.

CHAPTER 3

A Cache Layer Implementation for Spatial Data

The previous chapter presents different design variants for our distributed spatial cache and proposes a general software architecture composing multiple layers.

This chapter proposes implementation concepts for the *cache layer* of the general software architecture. This layer organizes the data cache of a single cache server. The concepts presented here are based upon a previously published work [Lübbe et al., 2011] using geographic parcels in spatially partitioned data space.

In the following, Section 3.1 introduces the required data structures for parcel-based caching. Section 3.2 describes the general mode of operation. Section 3.3 details on the implementation of different caching strategies. The basic approach is assessed in Section 3.4. Furthermore, Section 3.5

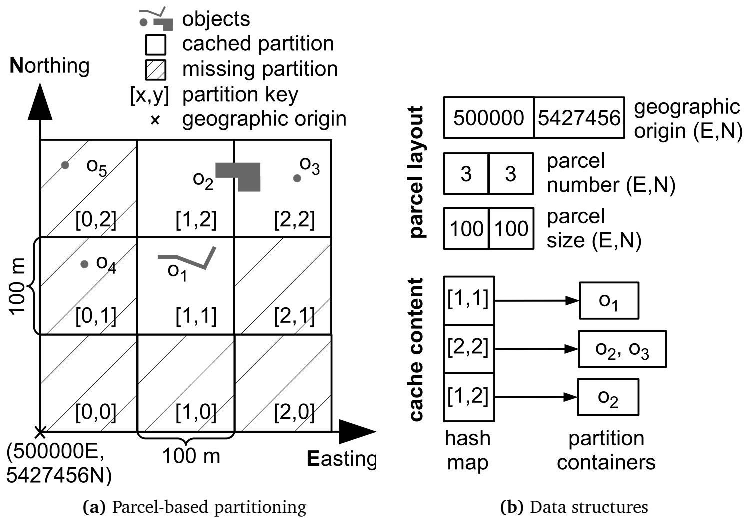
proposes an extension to our basic approach for non-spatial attribute dimensions. Finally, Section 3.6 summarizes the chapter and gives an outlook on the next chapter.

3.1 Spatial Partitions as Basic Units for Caching

In Chapter 2.2.2 we argue in favor of partitions as the prime choice granularity level for caching. Our key argument is that partitions provide considerable flexibility at reasonable overhead for the cache management. Using appropriate transformation methods, geographic coordinates can be converted into a two-dimensional Euclidean space in which the coordinates of a point are expressed as easting and northing. Here, easting refers to the distance measured in eastward direction, whereas northing refers to the northward distance. In such a geographic Euclidean space, partitioning is straightforward.

In this way, we partition geographic data into parcels forming a grid. Each partition corresponds to a spatial parcel and contains data objects whose geographic extent intersects with that parcel. The spatial extent of objects can resemble arbitrary geometric features, such as points, lines or even polygons. If an object overlaps multiple parcels it is contained in each partition whose parcel intersects with the object's extent. The partitions are labeled with a unique key $[x, y]$ where x and y determine the position of the corresponding parcel in the grid. Figure 3.1a depicts an example for such a parcel-based partitioning layout. In the figure, the spatial objects o_1, o_2 and o_3 are already cached while o_4 and o_5 are contained by partitions which are missing in the cache. o_2 is contained by partition $[1, 2]$ as well as $[2, 2]$; the other objects are contained by only one partition each.

Figure 3.1b shows the required data structures to represent the parcel-based partitioning layout. The cache server stores information about the parcel layout, such as the grids geographic origin, the number of parcels and the parcel size in easting and northing direction. With this information, we can easily derive the geographic extent of the corresponding parcel from a given partition key [x,y]. Thus, for any given geographic target region, the



(a) Parcel-based partitioning

(b) Data structures

Figure 3.1: Spatial partitions as caching units

cache server is able to determine the set of partitions whose parcels intersect the target region. In addition to that, the objects of a certain partition are stored in dedicated partition containers and are hashed according to the corresponding partition key. Each partition must be complete, i.e. all objects intersecting the corresponding parcel are stored in the respective partition container. Partitions which are currently not cached possess no entry in the hash map and thus require no memory at all.

The objects can be stored in the partition containers in in-place or referenced manner, as indicated by Figure 3.2. In the in-place mode, the objects are stored directly inside the partition container. This may cause the duplicate storage of an object, when it is contained by multiple partitions, as shown in Figure 3.2a. To avoid duplicate storage, the objects can be stored in a referenced manner, as shown by Figure 3.2a. In the referenced storage mode the partition containers contain only pointer which reference the respective objects.

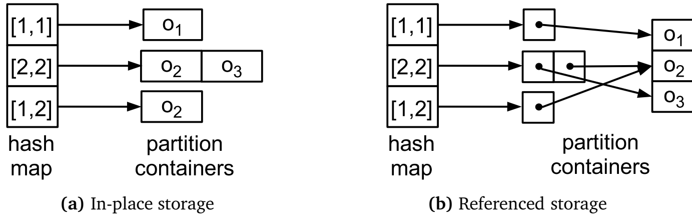


Figure 3.2: Storage modes for the partition containers

3.2 Operation Mode

To process a request on top of the previously described spatially partitioned data cache, the following steps have to be accomplished:

1. **Cache search:** For a given target region of a range query, determine the set of partition keys $K = \{[x_1, y_1], [x_2, y_2], ..., [x_k, y_k]\}$ of intersecting parcels. For each key $[x_i, y_i] \in K$, check if the hash map contains an entry for the corresponding partition.
   a) if yes, then add all objects of partition $[x_i, y_i]$ to a temporary data structure containing the local result.
   b) if not, add the corresponding geographic region of $[x_i, y_i]$ to the back-end request which delineates the geographic region of all missing parcels.
2. **Load missing data:** Retrieve all missing partitions identified in step 1b) from the data back-end and keep all retrieved objects in a temporary data structure containing the back-end result.
3. **Insert loaded data:** Insert the partitions retrieved in step 2) into the hash map. If the current cache capacity is reached, replace existing partitions according to a dedicated caching strategy (cf. Section 3.3).
4. **Result preparation:** Merge the local result with the back-end result. Due to the inaccurate data identification abilities of partitions, the result may still contain dispensable data. Therefore, remove each

object which does not satisfy the original query. In addition, remove duplicates which are caused by objects overlapping multiple parcels.

3.3 Caching Strategies

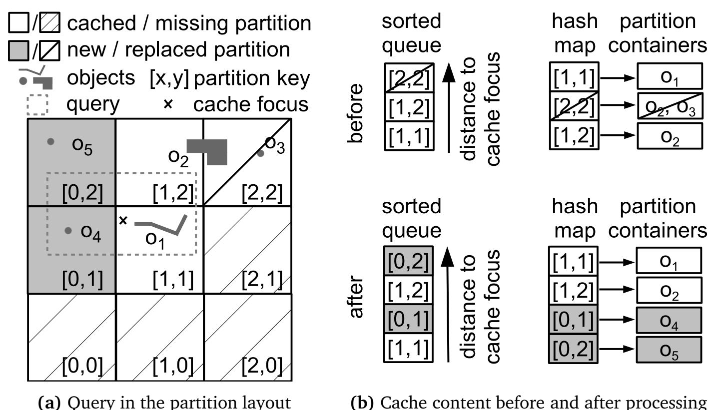
To keep only the data fraction in the cache which is expected to be required for future requests, different strategies are conceivable. Implementing well-known caching strategies, such as LRU, on top of a partitioned-based caching architecture is possible. The general procedure for this is sketched in the following Section 3.3.1. Beyond the traditional variants, we propose an alternative strategy specifically suited for global cache content control in a distributed system in Section 3.3.2.

3.3.1 Traditional Caching Strategies

With spatial partitions as basic units for caching, traditional caching strategies can be easily implemented. In an LRU implementation, for instance, the partition keys are inserted into a queue which is sorted by the recency of their usage. Replacement candidates of seldom used partitions can then be found at the tail of the queue. However, traditional replacement schemes, such as LRU, cannot directly be applied in a distributed setting where content is distributed among multiple servers. It also requires additional mechanisms to distribute the workload among the cache servers in a way so that access locality is preserved. In our first approach presented in [Lübbe et al., 2011], we achieved this using a hierarchical spatial partitioning scheme to distribute content and workload among multiple independent LRU cache servers. One key observation we made in this partitioned LRU setting, is that the approach lacks the required flexibility to support effective real-time load balancing (cf. Section 4.1.2).

3.3.2 Focused Caching

To increase the flexibility of content allocation for the distributed load balancing process, we proposed *focused caching* as an alternative replacement



(a) Query in the partition layout **(b)** Cache content before and after processing

Figure 3.3: Query processing using *focused caching*.

strategy in [Lübbe et al., 2012]. The focused caching strategy replaces those partitions first which have the highest Euclidean distance to a dedicated point in space, denoted as the *cache focus* of the server. Thus, a cache server preferably caches data near its cache focus, as sketched in Figure 3.3a. It is implemented using a queue containing the keys of all cached partitions sorted by the Euclidean distance to the cache focus. The freely assignable cache focuses enable a very flexible mapping of content to servers. Thus, it is the ideal basis for our real-time load balancing approaches presented in Section 4.

A simple example is sketched in Figure 3.3. The given query intersects with the partitions $[0,2]$, $[1,2]$, $[0,1]$ and $[1,1]$, as it can be observed in Figure 3.3a. As the partitions $[1,1]$ and $[1,2]$ are already cached, the objects o_1 and o_2 can be added directly to the temporary local result. The missing partitions $[0,2]$ and $[0,1]$ are then retrieved from the back-end. The contained objects o_4 and o_5 are added to the temporary back-end result. As the cache is only capable to store 4 objects in the example, existing partitions have to be replaced in order to free up space for new data. As $[2,2]$ is the partition which is furthest away from the cache focus position,

it is chosen as replacement candidate and removed from the cache. Then, the objects retrieved from the back-end and the corresponding partitions keys are added to the data cache, as depicted in Figure 3.3b. Finally, the local and the back-end result are merged and the objects o_2 and o_5 are removed from the final result, as they do not intersect with the original target region.

3.4 Assessment

After having sketched the general concepts of our parcel-based caching approach, this section assesses the performance relevant characteristics of our implementation. The performance is mainly influenced by the number of involved partitions which directly depends on the size of the parcels: i.e., a small parcel size leads to a high number of partitions whereas big parcels reduce the number of required partitions. In our consideration, we assess several performance relevant criteria in respect to the parcel size.

In the following, Section 3.4.1 to Section 3.4.3 assess the time complexity, the space complexity and the accuracy of data identification for varying parcel sizes in a concrete application scenario. Finally, Section 3.4.4 summarizes the findings and gives a recommendation for reasonable parcel size in the context of our application scenario.

3.4.1 Space Complexity

In this section, we estimate the additional space overhead which comes on top of the cached payload data. In its basic version, our approach requires a hash table which contains key entries of cached partitions. A key entry consists out of three 32 bit long fields: two integers define the partition key and one pointer gives access to the corresponding partition container. Moreover, we assume that each object is contained by at most one partition, i.e., objects do not overlap with parcel borders. With these prerequisites the additional space overhead for the hash map can be estimated as $32 \times 3 \times p$

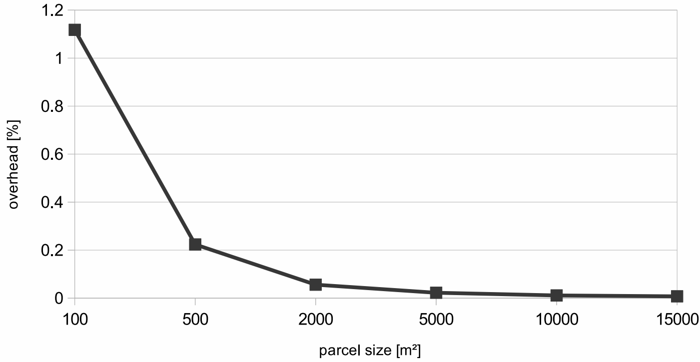


Figure 3.4: Space overhead for caching of a concrete application scenario

bits, when p is the number of partitions.[1] If A is the total area covered by the parcel layout, the parcel size can be defined as $a = \frac{A}{p}$. Thus, the space overhead in respect to the parcel size a can be defined as $s(a) = 32 \times 3 \times \frac{A}{a}$. In relation to the payload size R, the proportion of the additional overhead is $\frac{s(a)}{R}$.

We use a concrete example to illustrate the estimation formula. We assume that the parcel layout covers a total area of $A = 100\,\text{km}^2$. This easily requires $R = 1$ GByte of memory in particular when it includes large binary data, such as geo-referenced images. Figure 3.4 depicts the relative space overhead required to cache the complete data extract for varying parcel sizes. Even for a small parcel size of 100 m^2 corresponding to square $10 \times 10\,\text{m}$ parcels, it can be observed that the overhead generally stays under 1.2 % of the payload data.

The above estimation does not consider objects which cross parcel borders and are therefore contained by multiple partitions. When the objects

[1] We assume that the hash table is completely filled. In practice, hash tables often contain a certain percentage of empty buckets so that collisions are avoided and lookup performance is increased. Typically the percentage of empty buckets stays below 25 % of the stored entries.

are stored in a materialized way inside the partition containers, the duplicate storage of border objects induces significant overhead, especially when the average object size is high and the parcels are very small. Therefore, the partition containers store contained objects in a referenced manner, i.e., they maintain pointers to the contained objects instead of storing them in-place (cf. Figure 3.2b). Thus, duplicate storage of objects is avoided. If we assume an average object size of 50 KByte, our 1 GByte data set contains approximately 20000 objects. With a pointer size of 32 bit, the total overhead for storing the 20000 references is approximately 20000×32 bit $\approx 20000 \times 4$ Byte $= 80$ KByte. Hence, even under the pessimistic assumption that each object crosses a parcel border, the overhead for referenced storage is still only 160 KByte. In other words, our estimation depicted by Figure 3.4 is not significantly affected by overlapping objects due to a referenced storage.

3.4.2 Time Complexity

The processing time of an arbitrary process strongly depends on the number operations required to execute this process. In this section, we assess the number of operations that a cache server has to execute in order to process a request. The number of operations depends on the number of partitions and thus indirectly on the actual parcel size. In the worst case, the time required to process a request corresponds to $3 \times p + 2 \times p \times N$, whereas p is the number of partitions and N is the average number of objects per partition. The first summand of this formula encapsulates three steps which operate on partition level and therefore only require p operations each: the identification of intersecting parcels (cf. step 1, Section 3.2), the hash table lookup (cf. step 1a and 1b, Section 3.2) and the possible replacement of cached partitions (cf. step 3, Section 3.2). The second summand delineates two steps which operate on object level and therefore require $p \times N$ operations each: the deserialization of result objects during load (cf. step 2, Section 3.2) and the eventual result preparation step (cf. step 4, Section 3.2). If A is the total area covered by the parcel layout, the parcel size can be defined as $a = \frac{A}{p}$.

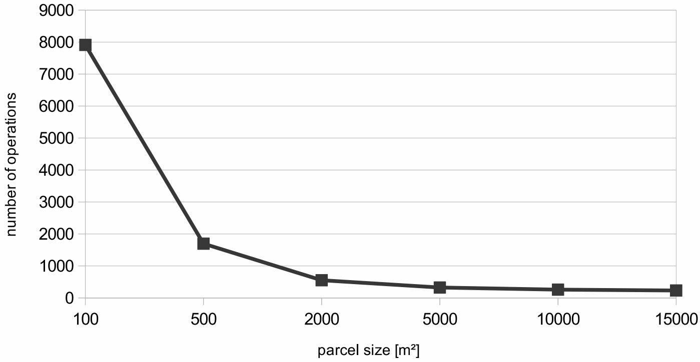


Figure 3.5: Number of operations necessary to process a request

Thus, the worst case time complexity in respect to the parcel size a can be defined as $t(a) = 3 \times \frac{A}{a} + 2 \times \frac{A}{a} \times N$.

We examine the time complexity using a concrete example. Assuming range queries with a target region of $500 \times 500\,m$, we can deduce the maximum number of involved partitions and the average number of objects per partition for varying parcel sizes. Using the formula given above, we calculated the maximal number of operations required to process the range query for various parcel sizes, as depicted in Figure 3.5. One can observe that the number of required operations increases for small parcel sizes, as the number of involved partitions is high at this granularity grade.

3.4.3 Data Identification Accuracy

The granularity grade of parcel-wise partitioning affects accuracy of data identification. Fine grained parcels allows precise identification of missing data while an increasing parcel size degrades the accuracy of back-end requests causing additional data transfers during cache loading. In this section, we quantify the amount of these additional data transfers. For this

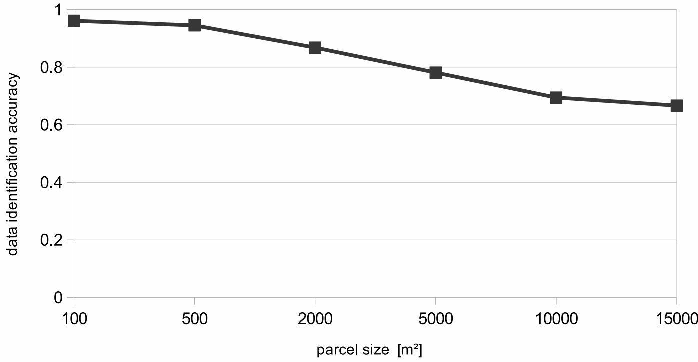


Figure 3.6: Data identification accuracy

purpose, we consider a concrete scenario. In this scenario, we assume that the cache is empty. To process a given query, the cache server requests all parcels from the back-end which intersect the target region of the query. Because of the inaccuracy of parcel-based data identification, the cache server typically needs to retrieve more from the back-end than actually required to process the given query. The amount of this additional data transfer depends on the parcel size: Smaller parcels minimize the additional transfer while bigger parcels increase the additional transfer. As shown in Figure 3.7, a square target area of a range query area overlaps with at most one line of parcels at each side of the target area. Thus, for a given square target area A_Q of a range query, the maximum area of the back-end request in respect to the parcel size a can be estimated as $a_B(a) = (\sqrt{A_Q} + 2 \times \sqrt{a})^2$. Thus, we can determine the ratio of the actual query area and the area of the back-end request: $i(a) = \frac{A_Q}{a_B(a)}$. Assuming that objects are uniformly distributed in space, this function is proportional to the volume of additional data transfer. Therefore, we use it to characterize the data identification accuracy.

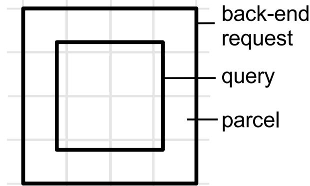


Figure 3.7: Back-end request for a square query region

We determine the data identification accuracy in the context of a concrete example. Assuming square range queries with $A_Q = 500\,\text{m} \times 500\,\text{m} = 25000\,\text{m}^2$, we can calculate concrete values for $i(a)$ in respect to various parcel sizes, as depicted in Figure 3.6. As it can be observed, the accuracy generally stays above 60 % for all examined parcel sizes. A parcel of 100 m^2 achieves an accuracy of 96 %, 500 m^2 parcels still achieve 95 %.

3.4.4 Summary

As it can be observed, the space and time overhead increases significantly for shrinking parcel sizes, in particular for parcel sizes below 500 m^2. At the same time, the increase of the data identification accuracy is insignificant for parcel sizes below 500 m^2 (only 1 % increase when comparing 100 m^2 and 500 m^2 parcels). We regard the additional overhead not as justifiable considering the small increase of data identification accuracy. Therefore, we generally recommend parcel sizes above 500 m^2. The appropriate value depends on the requirements of the application field. For example, in our evaluation scenario used in the experiments evaluating the cluster layer (cf. Chapter 4), we considered a data identification accuracy between 80 % and 90 % as sufficient. This approximately corresponds to parcel sizes ranging from 3000 m^2 to 4500 m^2.

3.5 Beyond Spatial: Supporting Alternative Attribute Dimensions

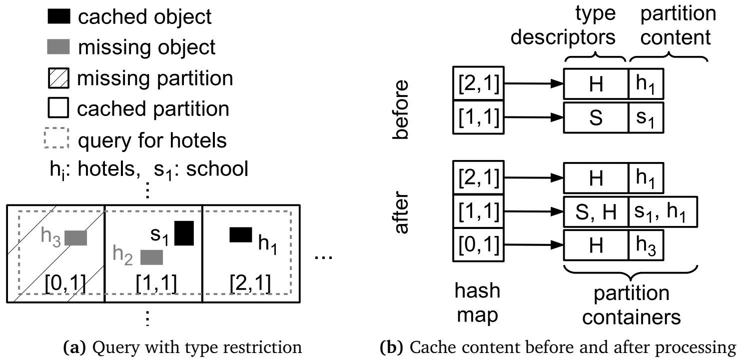
The caching architecture presented above focuses on the spatial dimension, as location often is the primary selection criterion in our application field. It is most adequate for queries which reference complete data extracts of geographic regions. However, queries including additional attribute dimensions may decrease the data identification accuracy. For instance, a digital hotel finder, may typically reference data objects with the type "hotel" in certain geographic areas. The parcel-wise loading of all object types fills the cache with a lot of data the hotel finder cannot do anything with. In this case, a more selective loading mechanism is required which is able to retrieve objects of a specific type in geographic area.

In this section, we first extend our basic parcel-based caching mechanism to support type attributes. Subsequently, we discuss the implications of extending our concept to arbitrary attribute dimensions.

3.5.1 Type-enabled Parcel Caching

To extend our concept, we added an additional field, denoted as *type descriptor* to the partition container. The type descriptor semantically describes the data in the respective partition. It contains a list of object types which are currently cached in a particular partition. The partition must be complete in respect to the given types in the type descriptor, i.e., if the type descriptor contains a certain type, all objects possessing that particular type and intersecting the partition's geographic parcel must be contained by the corresponding partition container. To process range queries containing a type restriction, the cache lookup has to be modified (cf. step 1, Section 3.2): A partition then only qualifies as a cache hit, if the corresponding type list contains the requested key. If this is not the case, the objects of that type from that particular partition have to be loaded from the back-end.

Figure 3.8 depicts an example for the cache management in case of typed region queries. Suppose our cache already contains the school object s_1 in partition $[1,1]$, as well as the hotel object h_1 in partition $[2,1]$. The objects



(a) Query with type restriction

(b) Cache content before and after processing

Figure 3.8: Query processing in the type extended parcel-based cache.

h_2 and h_3 are not in the cache. Now a client requests all hotels in a region intersecting the parcels $[0,1]$, $[1,1]$, $[2,1]$ as indicated in Figure 3.8a. The content of partition $[2,1]$ can be used to answer the request, as it contains all hotels intersecting the corresponding parcel. However, the hotels intersecting parcel $[1,1]$ have to be loaded from the back-end, as the hotel type is not contained by the partition. Finally, h_3 is also loaded from the back-end as the hash map does not contain an entry for the intersecting parcel $[0,1]$. The corresponding objects and type descriptors are added to the cache and the final result set is obtained by merging the local and the back-end result.

3.5.2 Discussion

Our type extension introduces additional space overhead for storing the type descriptors: in the worst case $t \times p \times 32$ if t is the number of types p the number of partitions and types are encoded in 32 bit long fields. In addition, the time complexity increases as more operations must be executed during processing: the type descriptor has to be checked during cache lookup (cf. step 2, Section 3.2) and it has to be updated during load (cf. step 3, Sec-

tion 3.2). This additional overhead is proportional to the number of types stored in the cache. As realistic data sets only contain a limited number of object types (typically less than 100), this overhead is manageable.

Principally, our approach can be extended to further attribute dimensions. In the case of the digital hotel finder, for example, another attribute dimension, describing the price of single bedrooms can be useful. To include this, we propose to partition the price dimension into disjunct price categories, e.g., low-price (less than 50€), mid-price (from 50€ to 100€) and high-price (above 100€). The additional overhead depends on the number of included price categories. In our example of three price categories, the overhead is manageable. In general, it has to be examined whether the overhead introduced by the additional attribute dimension is justified by the gain of data identification accuracy.

3.6 Summary and Outlook

This chapter proposes an implementation concept for the cache layer of the software architecture presented in Chapter 2.4. The implementation bases on spatial partitions as basic units for caching. By using partitions, we are able to achieve high data identification accuracy while keeping the space and time overhead for the cache management within reasonable bounds. On top of the spatial partitions, we propose a concept to integrate alternative attribute dimensions.

The following chapter deals with the implementation of the cluster layer of our proposed software architecture (cf. Section 2.4). The cluster layer combines multiple cache servers to form a distributed spatial cache. For this purpose, the cluster layer uses an implementation of the cache layer, such as the one previously described in this chapter.

CHAPTER 4

Implementation Concepts for the Cluster Layer

In Section 2.4, we propose a software architecture comprising multiple software layers. The previous chapter details the cache layer which organizes the cache on a single cache server. This chapter focuses on the cluster layer which combines multiple cache servers in order to form a distributed spatial cache.

A key challenge in this context is to distribute the data content among the participating cache servers in an adequate manner. The most obvious approach for this is to separate the data space into several partitions and assign each cache server a dedicated partition in the global partition layout. Section 4.1 describes this approach. Another option for content arrangement among the cache servers is the use of the focused caching for cache replacement, as illustrated in Section 4.2. Section 4.3 discusses our results.

Finally, Section 4.4 summarizes this chapter and outlines the remainder of this work.

4.1 Partition-based Content Arrangement

Partitions form the basis in many content delivery networks, such as ASPEN [Wang et al., 2005] or GeoGrid [Zhang et al., 2007] or CAN [Ratnasamy et al., 2001]. In these networks the data space is partitioned into distinct parcels. Each server stores the content of a certain parcel in this partitioning layout. When a client requests the content of a certain partition, the responsible server is contacted to process the request. At a first glance, this concept seems to be directly transferable to distributed caching, except that the cache content is not stored permanently (in contrast to traditional content delivery networks), but is replaced regularly due to limited cache storage capacity. Whether partition-based content arrangement is also viable for distributed caching is a legitimate question. The objective of this section is to give an answer to this question.

For this purpose, we extended a popular content delivery network to support caching. Section 4.1.1, describes the characteristics of this cluster layer implementation. The subsequent Section 4.1.2 discusses the partition-based approach in the context of the previously asked question.

4.1.1 A Parcel-based Cluster Layer Implementation

Parcel-wise partitioning forms the basis of a very popular content delivery network with the name Content Addressable Network (CAN) as published in [Ratnasamy et al., 2001]. CAN is a distributed mechanism providing a hash table-like functionality. Its design is based on a d-dimensional Cartesian coordinate space which is partitioned among the participating servers during run-time. Each server in the network is responsible for a distinct zone in the coordinate space. By storing the IP addresses of servers with adjoining zones, the servers form an overlay of logical links. Thus, a server's routing table contains the set of immediate neighbors in the coordinate

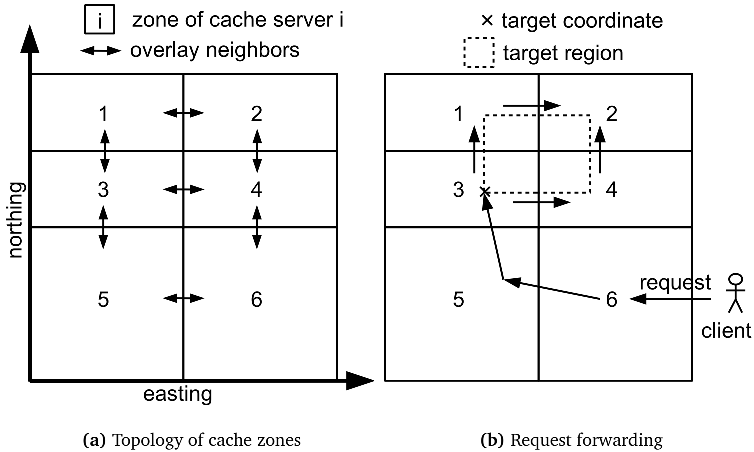
space and a data structure describing the extent of each neighbors zones. The overlay network is used to store key-value pairs in which keys are coordinates in the d-dimensional coordinate space. Each server stores only those key-value pairs with key coordinates located in its zone. To retrieve a value with a certain key, the request is routed through the network. Thereby a server uses its routing table to send the request to the neighbor whose zone has the smallest distance to the target coordinate until it reaches the server owning that coordinate.

Inspired by these concepts, we designed and implemented a distributed spatial cache in a previous work [Lübbe et al., 2011]. In several aspects our implementation goes beyond the basic concepts of CAN. We describe the major operational differences in the sections 4.1.1.1 to 4.1.1.6. Finally, we evaluate this cluster layer implementation in Section 4.1.1.6.

4.1.1.1 Geographic Content Arrangement

Using common map transformation methods, the Cartesian coordinate space of CAN can be transformed to geographic coordinates. Thus, CAN can be adapted to store geographic data. In the parcel-wise partitioning layout, each cache server is in charge of an individual geographic cache zone. All objects cached by a certain cache server geographically intersect with its cache zone. As a consequence, objects whose geographic extents overlap a certain zone border are possibly cached by all the cache servers sharing that border. This property is essential to efficiently handle range queries which possibly reference such border objects.

On top of the parcel-wise partitioning layout, a geographic overlay topology is built as depicted in Figure 4.1a. The link structure in the overlay corresponds to the geographic topology of the cache zones, i.e. neighbors in the overlay are also neighbors in the geographic coordinate space. This property is required to efficiently process spatial queries as explained in the following section.



(a) Topology of cache zones

(b) Request forwarding

Figure 4.1: A geographically partitioned distributed spatial cache

4.1.1.2 Query Processing

Clients send their requests to the cluster running the distributed spatial cache. Using a specific cluster edge technology (cf. Section 2.3.3), these requests are distributed among the cache servers. As the cluster edge is typically not aware of the cache server's current cache content, the requests may arrive at an arbitrary cache server. In consequence, the distributed spatial cache must implement a mechanism which identifies those cache servers that can possibly provide results to a certain request.

In our case, all cache servers whose cache zones intersect with the target region can possibly contribute to the result. To find those appropriate cache servers, we extended the basic routing mechanism of CAN which is able to route messages to a certain target coordinate in its coordinate space. As target coordinate, we choose an arbitrary coordinate of the request's target region. Then the distance-based forwarding mechanism of CAN routes the request to the cache server owning this coordinate. This server generates a

unique process ID which can be used to distinguish parallel query processes and duplicates that may arise during the routing process. Thereafter, the destination server sends the request to all neighbors whose cache zones also intersect with the target region. This process is possibly repeated to also reach the cache servers which are not direct neighbors of the destination server. Subsequently, all identified cache servers with intersecting cache zones search their local data caches while possibly loading missing data from the back-end. Finally, the result is merged and is send back to the client.

Figure 4.1b depicts the routing path of a range query in the geographic coordinate space. Server 6 initially receives the range query and extracts a target coordinate from target region. In our example the target coordinate is the lower left corner of the target region. The servers (i.e., first server 6 and then server 5) use the distance-based forwarding mechanism of CAN to route the request to the cache server owning the target coordinate which is server 3 in our case. In the example, the target region overlaps with multiple cache zones. Therefore, server 3 forwards it to all neighbors intersecting the query region, i.e. server 1 and server 4 in our example. Server 1 and server 4 both forward the request to server 2, as this server is not an immediate neighbor of server 3 but still overlaps with the query region. Using the unique process ID, server 2 is able to identify and ignore the duplicate requests send by servers 1 and server 4. In this way, our routing extension of CAN is able to identify all affected cache servers even for complex geographic query regions. Each identified cache server processes the part of the target region intersecting with its cache zone and sends the partial results back to server 3. Then, server 3 merges the partial results. The final result set is returned to the client which initially issued the request.

4.1.1.3 Adaptive Partition Split

As explained in the previous section, requests are routed to the cache server whose cache zones intersect with the target regions. In consequence, cache servers covering highly requested regions will receive significantly more requests than the rest of the cache servers. This leads to load imbalances

between the cache servers. In the worst case, it causes high response times, when certain cache servers are overloaded.

In this section, we introduce a partitioning scheme which is aware of such load imbalances. This load-aware partitioning mechanism can be used for advanced load balancing and scaling mechanisms. In the following, we first describe the load-aware partitioning mechanism and subsequently propose a load-balancing and scaling mechanism on top of our partitioning mechanism.

In our load-aware partitioning scheme, the partitions are created dynamically, whenever a new cache server joins the distributed spatial cache. The first cache server obtains the complete address space as its cache zone. Subsequently joining cache servers send a join request to an arbitrary cache server which is already part of the distributed spatial cache.

Whenever a cache server in the distributed spatial cache receives a join request, an existing cache zone has to be split. Finding an adequate split candidate is essential for efficiently dealing with load variations, as it affects several performance relevant criteria of the distributed cache:

- **Hit-rate:** The splitting affects the size of the cache zones. When data is spread evenly the data volume of a cache zone is proportional to its size. Thus, splitting typically reduces the amount of data a certain cache server is responsible for. The reduction of the cache zone size, enables a cache server to keep a greater portion of the cache zone in its limited cache storage. With the increased cache zone coverage, the probability of an object being cached increases and thus the hit-rate improves.
- **Resource utilization:** Splitting can affect the filling level of the cache servers. Excessive splitting may cause small cache zones containing little or no data and thus may lead to bad resource utilization.
- **Workload:** Splitting also influences the workload distribution among the cache servers. Reducing the cache zone size of a certain cache server typically also reduces the number of queries it has to handle. An adequate splitting increases the overall throughput of the system, as load is shared between cache servers. When the average query

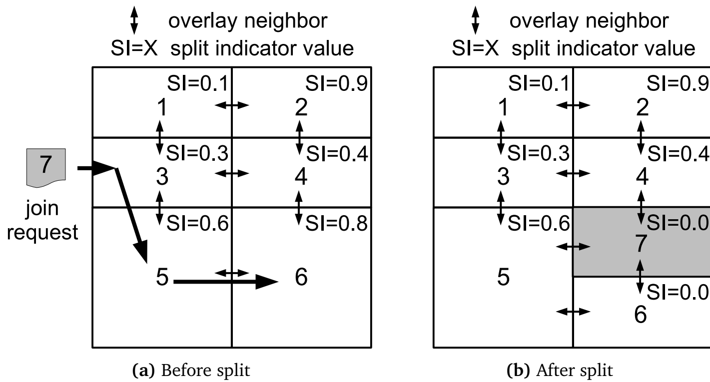
region size becomes larger than the cache zones, additional overhead incurs, as multiple cache zones overlap with the query region.

- **Routing overhead:** The splitting increases the number of servers in the distributed cache. This enlarges the number of hops for routing messages in the overlay topology.

Some of the above mentioned criteria are conflicting, e.g., a split decreases the average workload of the cache servers, but at the same time it increases the routing effort. In its essence, finding a proper split candidate is a multi-dimensional optimization problem. Finding an optimal solution requires a detailed and up-to-date insight into the run-time statistics of each cache server. As these statistics are dispersed among different cache servers, the number of messages required to obtain such a detailed overview of the complete system is proportional to the number of cache servers. As this would induce a lot additional network traffic, we pursue a heuristic approach.

In our case, the decision whether to split a certain cache zone or not must be based on local knowledge. For this purpose, the routing table of each cache server contains an additional entry, denoted as the *split indicator*. The *split indicator* is a quantity which characterizes the cache server's run-time situation, such as the current workload, the current resource utilization or others. The cache servers periodically exchange up-to-date split indicator values with their neighbors using dedicated update messages. Knowing the current split indicators of the overlay neighbors, enables a cache server to compare different split candidates. With this information a cache server is able to forward the join request to the neighbor with the highest split indicator. This process is repeated until no neighbor with a higher split indicator can be found.

Figure 4.2 depicts an example of the split process. In the example, the cache server 3 receives a join request for the new cache server 7. The request is routed to cache server 6 which eventually splits its cache zone in order to make space for the new cache server. Our heuristic split mechanism does not guarantee to find the best split candidate at a global scale, but identifies local optima for splitting. However, the cache server with the



(a) Before split

(b) After split

Figure 4.2: Adaptive partitioning

highest split indicator (i.e., cache server 2 in our example) will eventually be split, when the join requests occur uniformly distributed.

Our adaptive split mechanism can serve as a basis for an automated cluster maintenance, load balancing and scaling process. In the following, we sketch the general idea for implementing these processes on top of our split mechanism.

4.1.1.4 A Proposal for a Fully Automated Cluster Maintenance Process

Constant maintenance of our distributed spatial cache is necessary for two reasons: A cache server can either leave the distributed spatial cache intentionally or by a crash. In case of an explicit departure, a cache server sends a message to each of its neighbors informing them about its intention to leave. A crashed cache server is detected by the neighbors through the absence of heartbeat messages which also propagate split indicator values. In both cases (i.e., explicit departure or crash of a running cache server) another cache server has to take over the leaving cache server's job. To replace a crashed or explicitly removed server, [Ratnasamy et al., 2001] propose a

basic replacement algorithm which does not consider the current run-time situation of the servers. In contrast, we propose to include this information by exploiting the split indicator values. Thus, a cache server departure can be handled as follows:

Once a cache server fails or explicitly leaves the overlay, two actions are triggered: First, an immediate overlay neighbor helps out and temporally operates the leaving server's cache zone alongside with its own zone. In contrast to CAN which uses the zone size to determine the take-over neighbor, we propose to delegate the zone to the neighbor with the smallest split indicator, as this neighbor is the less congested one. The chosen neighbor then starts a reassignment process in the background to find a suitable replacement for the departed server. The replacement server can be obtained by merging the cache zones of two adjacent servers.

As the cache servers typically possess a limited capacity which is ideally fully utilized, a leaving server's data content probably cannot be taken over the other cache servers. For this reason, we abstain from transferring any data during a server departure. This is opposed to the CAN which assumes sufficient resources for data content take-over between servers.

4.1.1.5 Proposed Load-Balancing and Scaling Mechanisms

The previously described split mechanism can serve as basis for advanced load balancing and scaling mechanisms. In the following, we propose possible implementations on top of our adaptive split mechanism.

- **Proposed load balancing mechanism:** Load balancing can be effectively organized in a self-governed way by the cache servers. For this purpose, a cache server uses the split indicator value to locally measure the adequateness of its cache zone size in respect to the workload. If the split indicator falls below a defined threshold, the cache server initiates the departure process as explained in Section 4.1.1.4. Subsequently, the cache server sends a join request to one of his former neighbors. Through split indicator based forwarding, the server will then obtain a cache zone with a higher split indicator.

- **Proposed scaling mechanism:** A proper scaling decision should be based on a global scale indicator value that relates the current overall workload to the cluster's available computing capacity. If the value indicates an overload situation, the system should be scaled out, and vice-versa scaled-in when the system's capacity is over-sized in relation to the workload. A central resource manager component tracks run-time statistics of the cluster components and obtains the global scale indicator. To scale out, the resource manager sends a join request containing the address of a new cache server instance to an arbitrary server in the distributed cache. Using our split mechanism, the cache servers then autonomously find an appropriate split candidate and join the new server into the overlay. To scale-in, the resource manager sends dedicated retirement message to an arbitrary cache server. Exploiting the split indicator value, the request is forwarded to cache server with comparatively low value. This server then explicitly leaves the overlay using the departure procedure defined in Section 4.1.1.4.

4.1.1.6 Evaluation of the Parcel-based Implementation

We base our evaluation on PeerSim [Montresor and Jelasity, 2009]), which uses a discrete event-driven simulation model. In such a model, the simulated real world scenario is expressed as a sequence of events having a discrete time stamp. The simulator engine sequentially processes the events ordered by their time stamp. The discrete event-driven simulation model is suitable for the simulation of message passing behavior in overlay networks, as the messages can be easily expressed as timestamped events. For our evaluation, we created events for all involved messages of our distributed spatial cache and used the PeerSim framework to execute the simulation.

We evaluated the system performance through simulation of a real world scenario. In the following, we first illustrate the scenario and then discuss the results.

Simulation Scenario

Our simulation uses real world data of Berlin and its surroundings extracted from OpenStreetMap (OSM) [OpenStreetMap, 2011]. We converted the proprietary data format of OSM into a simplified format representing our data model defined in Section 2.1.1. Our data set includes a total number of 23,625 objects.

In our scenario, we simulate range queries of mobile clients, requesting data surrounding their current position. The target regions of the range queries are 500 × 500 m squares. For simulating the client positions, we consider two general behavioral patterns of users:

1. **Street Mobility:** In urban regions, a lot of people travel along streets. Many of them might request spatial data during their trip using smart phones or the on-board devices of their vehicles. To simulate such vehicle-like movement patterns, we base on the mobility simulation environment CANUMobiSim [Stepanov et al., 2003]. This tooling allows generating position traces of simulated vehicles movements along streets at various speeds.
2. **Hotspot:** In our application field data is often accessed in a non-uniform manner generating certain access hotspots for some geographic regions. Varying population density of the geographic regions may cause this behavior, or possibly differing popularity grades of certain locations or areas. To encapsulate such access behaviors, we simulate client positions according to Gaussian distributions.

We derived several concrete simulation configurations from the general behavioral patterns:

- **Mobility (n=1|10|100)**: In this configuration *n* clients move on streets while randomly changing their speed in between 30 km/h and 60 km/h. On their way, they issue range queries centered around their current position.
- **Hotspot (n=100):** In this configuration, we define five distinct hotspots in the center of Berlin. We simulate a total number of n=100 clients. At each hotspot a group of 20 clients issues range queries

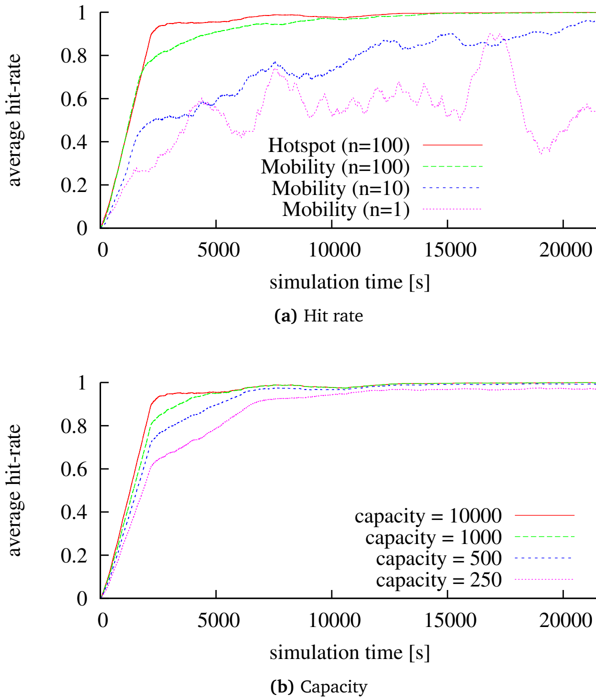
which are distributed around the hotspot according to a Gaussian distributions with a deviation of 1000 m.

We simulated our distributed spatial cache for a total amount of six hours simulation time. During the whole simulation time each client sends queries in equidistant time intervals of 30 s which results in 720 queries per client (i.e., for 100 clients this sums up to 72,000 queries in total). Our simulation is divided in two 3 hour phases: In the first phase the overlay is constructed, while cache servers subsequently join the overlay. During this process, every 108 seconds a join request is sent, until the overlay reaches the size of 100 cache servers. The cache zones are split according to two split modes: random split and adaptive split. In the *random split* mode, a random cache zone is chosen for splitting resulting in uniformly distributed splits. In the *adaptive split* mode, we find a suitable split candidate using the split indicator measure as described in Section 4.1.1.3. In this experiment the current workload of the cache servers constituted the split indicator. We measured the current workload of a cache server in the number of received requests per second. Throughout the whole first phase, the overlay processes the requests sent by the clients. In the second phase, the cache servers only process the client requests and the overlay structure does not change anymore.

Results

Based on the simulation scenario defined in the previous section, we examined several aspects of the system. In the following, we describe the most important aspects and discuss the main results of our evaluation. We refer the reader to our paper [Lübbe et al., 2011] for further details.

Examining the hit rate: The cache hit rate is a key performance indicator to characterize the effectiveness of caching. In our context, we define the hit rate of a request as the ratio of the number of objects retrieved from the cache and the total number of objects satisfying the request. A high hit rate signals that most of the work could be handled by the cache whereas a low hit rate indicates high involvement of the data back-end. In this experiment, each simulated cache server is capable to store 10000 objects. During the construction phase, the workload-based split scheme was used. In our



(a) Hit rate

(b) Capacity

Figure 4.3: Results of the simulation Part 1

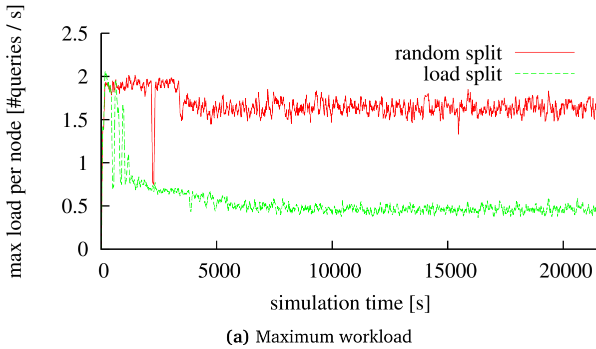
first experiment, we examined the cache hit-rate during different simulation configurations. Figure 4.3a depicts the moving average hit rate of a time window of 1500 s. In essence, the results demonstrate the impact of access locality between different clients. The single client ("n=1") cannot profit from inter-client locality and therefore exhibits the lowest overall hit rate. The occasional hit rate peaks are mainly caused by casually overlapping requests of this single client. In the other configurations inter-client locality plays an important role. Therefore, we observe higher hit rates for more clients. The hotspot configuration produces the highest hit rate. This

is due to the static position of the access hotspot which enables high cache exploitation.

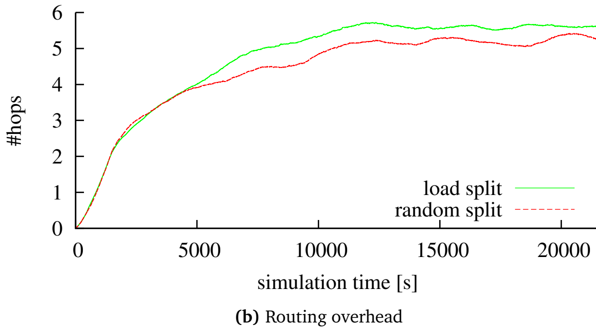
Impacts of server capacity: We conducted the previous experiment with sufficient cache capacity per server. In the next experiment, we gradually reduced the memory capacity of the cache servers in the "Hotspot (n=100)" configuration. Figure 4.3b visualizes the results. Even with a comparatively low cache capacity of 250 objects per server, the distributed cache overlay achieves hit rates above 90 %. Moreover, it can be observed that the hit-rate gradually improves during the overlay construction phase in the first half of the simulation. This is caused by the increasing amount of available cache memory when additional cache servers join the distributed spatial cache.

Examining the workload: To expose possible performance bottlenecks in the system, we inspected the workload of the cache servers in the simulation. In our context, we define the workload as queries per server and second. In this experiment, we simulated the "Hotspot (n=100)" configuration and measured the workload of all cache servers during the simulation. We picked the maximum workload value of all involved cache servers. Figure 4.4a visualizes the maximum workload for the random split mode and the adaptive split mode. It can be observed that the maximum workload is significantly reduced with the adaptive split mechanism, as the workload is distributed more effectively between the cache servers.

Routing Overhead: In the last experiment, we examine the effort of routing messages through the overlay. In this experiment, we simulated the "Hotspot (n=100)" configuration with a cache server capacity of 1000 objects per server. Figure 4.4b depicts the number of routing hops for the random split mode and the adaptive split mode. The adaptive split mechanism induces additional routing overhead. This is caused by an enlarged partition number in high load regions. Thus, a request has to be forwarded across more partitions until it reaches its final destination. However, it can be observed that the routing overhead of the adaptive split mode is comparatively low. Thus, the overhead is justified by the significantly better workload distribution observed in the previous experiment.



(a) Maximum workload



(b) Routing overhead

Figure 4.4: Results of the simulation Part 2

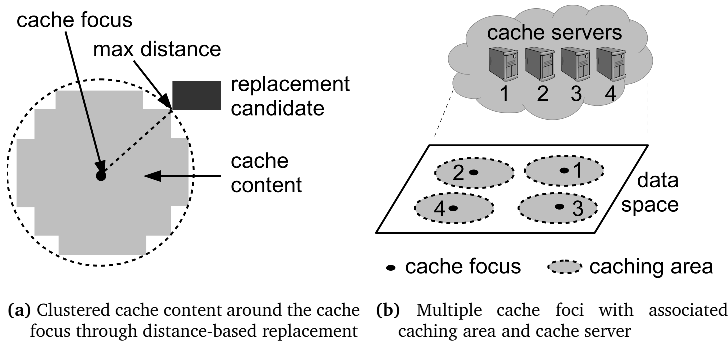
4.1.2 Discussion

We presented our extension to the partition-based content delivery network CAN that facilitates distributed caching of spatial data. Our main contribution is the adaptive split mechanism which can serve as a basis for advanced load balancing and scaling mechanisms. In our evaluation, we showed that our adaptive split mechanism provides a better workload distribution between the cache servers for certain non-uniform data access patterns.

Beside these positive results, the partition-based content distribution possesses inherent characteristics that can be disadvantageous in certain circumstances. In the following, we discuss two main characteristics which require a detailed examination:

1. **Partitions offer only moderate adaption potential:** The way how partitions are created is predetermined. In the case of CAN, partitions form rectangular parcels which may be split in two equally sized halfs when a new cache server joins the distributed spatial cache. Such a rectangular partition layout is only effective, if it distributes the workload evenly among the cache servers. This is not always the case for the data access patterns predominant in our application field. If the data access is not equally distributed among a partition, a split operation may cause uneven workload distribution among the newly created partitions. In the worst case, the complete workload is concentrated in only one of the newly created partitions after a split operation.
2. **Reorganization is expensive:** It is expensive to reorganize the partition layout. First, repartitioning causes content rearrangement between the cache servers. This may cause loss of cache content in particular when partitions are merged. This can temporarily decrease the cache hit-rate and increase the workload at the data back-end. Second, the reorganization of the distributed logical link structure includes an expensive search operation for split or merge candidates. This causes high message overhead, as the necessary messages may be forwarded over multiple hops.

For these reasons, we devise an alternative model to arrange the content among the cache servers. In the following section, we describe two possible implementation approaches for a distributed spatial cache which base on focused caching, as introduced in Section 3.3.2. With focused caching we are able to flexibly rearrange content among the cache servers and thus can adapt effectively to various data access patterns.



(a) Clustered cache content around the cache focus through distance-based replacement

(b) Multiple cache foci with associated caching area and cache server

Figure 4.5: Content arrangement based on focused caching

4.2 Content Arrangement Through Focused Caching

With our strategy of focused caching introduced in Section 3.3.2, a single cache server replaces the data which has the highest Euclidean distance to a dedicated point in space. This point, denotes the cache server's *cache focus*. Thus, data which is close to the cache focus is likely to stay in the cache during the replacement. Typically, this causes a cache server's cache content to be clustered around its cache focus, as visualized in Figure 4.5a. In this way, a cache server's preferred caching area can be influenced by allocating the server's cache focus in a certain position in space. We utilize this mechanism to arrange the overall cache content among several cache servers. Figure 4.5b depicts an example of several cache servers with distinct cache foci.

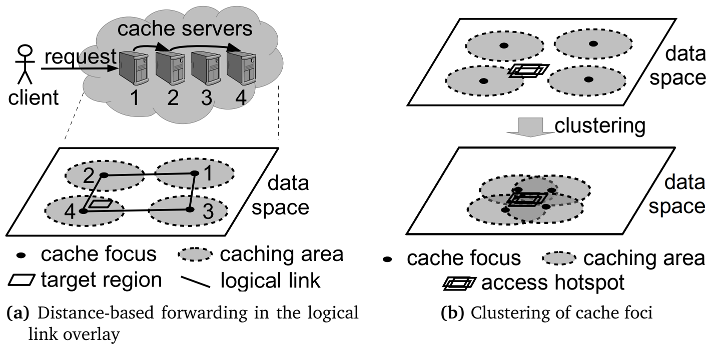
When data is requested from such a distributed content store, a suitable cache server for processing the request has to be found. Usually this should be a cache server which caches as much as possible of the requested data. The locations of the cache foci, can serve as an indication to find the most suitable cache server. Through our replacement strategy of focused caching (cf. Section 3.3.2) a cache server typically stores data referencing the area

around its cache focus. Thus, a server possessing a cache focus near the requested target region is likely to cache the requested data.

The task of finding the closest cache focus to the target region can be implemented in a centralized or distributed fashion. In a centralized approach, a central register keeps track of all the cache servers current cache focus positions. For a given request, the registry is searched for the cache server owning the closest cache focus. However a central register introduces a single point of failure and a performance bottleneck in a core task. Because of these disadvantages, we pursue a distributed approach as described in the following.

In our solution, each cache server keeps track of only a small subset of the available cache servers's cache focus positions. To organize these local cache focus mappings, we establish a link structure between the cache servers. In this link structure, two cache servers are connected via a link when they mutually know the positions of their cache foci and their network addresses. To ensure efficient retrieval of content, the link structure between the cache servers must resemble the topological distribution of cache content, i.e., servers with geographically related content should be connected via a link. Figure 4.6a depicts an example of such a geographic link structure. Utilizing the geographic arrangement of links, an incoming request can be forwarded in order to find a cache server possessing a cache focus close to the target region. In the greedy forwarding scheme, a cache server forwards an incoming request to the neighbor having the smallest distance to the target region until no neighbor with smaller distance is found. In a properly arranged geographic link structure, the forwarding process terminates at the cache server possessing the closest cache focus to the target region.

In such a network of cache servers, an adequate placement of cache foci in space determines the effectiveness of caching. Thereby, the density of cache foci in a certain region determines the amount of available resources to cache and process data of that region. For this reason, it is essential to position the cache foci of available cache servers in an adequate manner, i.e., in a way that matches the current demand. Thus, in regions with highly demanded data the density of cache foci should increase and vice versa



(a) Distance-based forwarding in the logical link overlay

(b) Clustering of cache foci

Figure 4.6: Data access in the distributed spatial cache

decrease in unpopular areas. Figure 4.6b shows an example of an access hotspot in the data space. The upper part of the figure shows a widely dispersed placement of cache foci while the lower part depicts a clustered cache focus placement around the access hotspot. The clustering of cache foci causes the cache server's caching areas to overlap to a great extent. Thus, a request with a target region within the overlapping area can be processed by multiple cache servers as the requested data is likely to be available on each of these servers.

To adequately allocate the cache foci in space, we introduce two mechanisms in the following. The first is based on the physical model of a particle-spring system to allocate the cache focus positions (see Section 4.2.1). The second bases upon the probabilistic distribution of cache foci via probability density functions (see Section 4.2.2). In Section 4.2.3, we examine the scalability of both approaches.

4.2.1 A Particle-Spring-based Implementation

In [Lübbe et al., 2012], we rely upon the physical model of a particle-spring system to position the cache foci in the space. In this model, massless particles are interconnected by springs. In physics, a constant factor k charac-

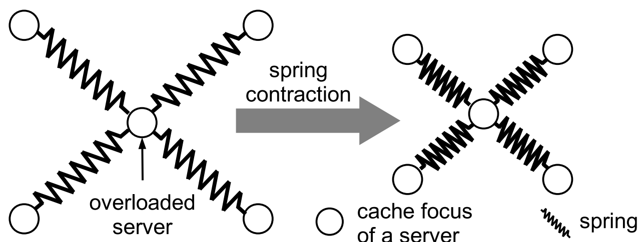


Figure 4.7: Load balancing through spring contraction.

terizes the spring's stiffness and x resembles its extension. In agreement with Hooke's law (as reported by [Petroski, 1996]), the spring's force F is proportional to its stiffness and extension, i.e., $F = k \cdot x$.

We transfer the physical model of a particle-spring system to our distributed cache overlay: A cache server resembles a particle whose position is defined by the server's cache focus. The cache focus of a cache server n is connected with the cache foci of neighboring cache servers n_i by springs. Thus, the Euclidean distance of the cache foci defines the corresponding spring's extension $x_i = |\overrightarrow{nn_i}|$.

To carry out our objective – i.e, to cluster the cache foci in regions with high load while decreasing their density otherwise – we contract the springs attached to overloaded cache servers and relax them otherwise. This causes a cache server with higher load to "pull" cache foci of the neighbors having less load closer to the position of its own cache focus, as visualized in Figure 4.7. The load of a single cache server n is characterized by a measure, which we denote as *gravity* g_n *of cache server* n. The cache server's gravity determines the amount the corresponding particle is going to pull on neighboring cache foci. Using the gravity of a cache server, we can define the spring constants of attached springs: For a spring i between the cache foci of a cache server n and its neighbor n_i, the sum of their gravities define the spring constant $k_i = g_n + g_{n_i}$. In general, a gravity measure can be defined in various ways. For instance, it could be simply derived from current run-time statistics, such as CPU usage, processing latency, current

cache occupancy and others. The exact definition of the measure depends on the specific application context. We provide an adequate definition of the gravity measure for our application field in Section 4.2.1.1.

4.2.1.1 Defining the Gravity of Cache Servers

As described in the previous section, we used a measure called *gravity* to characterize the load of a cache server. With this measure, we can define the spring stiffness of the particle-spring system. Transferring the server load to the stiffness of springs is a generic concept, as principally any measure can be included into the computation. In this section, we provide a reasonable gravity definition for our application context.

An obvious ingredient of the cache server's gravity is a characterization of its current workload, i.e, the amount of work a single cache server has been recently assigned to. Reasonable indicators to characterize the current workload are, for example, the current CPU usage, the number of incoming requests per second, or possibly the request processing latency, just to name a few. Another ingredient for the gravity is a measure for the amount of data within the area that a certain cache server is in charge of. If the cache server's memory capacity is too small to store all relevant data in the covered area, the measure should indicate an overload and vice-versa an underload situation.

Many existing load balancing mechanisms solely focus on either workload (e.g. [Wang et al., 2005]) or data skew (e.g. [Aberer et al., 2005]) without any considerations of the interactions between these two aspects. With respect to the cache server's memory constraints and the highly dynamic data access patterns of location-based applications, we consider both aspects (workload and data skew) as integral part of our load conception. For this reason, we provide a combination of both aspects as our measure to characterize the gravity of a cache server. In our previous work [Lübbe et al., 2012], we provide an example for such a fused measure. There, we define the gravity g_n of a cache server n as:

$$g_n = 1 + \beta \cdot (\rho_n \cdot \alpha + \omega_n \cdot (1 - \alpha))$$

Thereby, our gravity measure includes the following entities:

- ρ_n constitutes a measure for the data density in the region around the cache server's cache focus. We define it as $\frac{c_n}{a_n}$, where c_n denotes the number of cached objects and a_n the area covered by the cache server n. For our parcel-based cache, a_n is proportional to the number of cached partitions (cf. Figure 3.1).
- ω_n is an indicator characterizing the cache server's current workload. In our gravity measure, we use the number of incoming requests per second for the cache server n as an indicator for the current workload. We expect other indicators for workload such as CPU usage or processing latency to behave similarly.
- $0 \leq \alpha \leq 1$ is an application-defined parameter to adjust the weight between both aspects (workload and data density) and
- $0 \leq \beta < \infty$ is an application-defined parameter to scale the total value of the gravity measure.

It can be observed, that the gravity measure increases for cache servers in regions with high data density ρ_n. A higher gravity value increases the stiffness of springs and causes them to pull with more force. In consequence, the springs contract in regions with high data density. This increases the cache focus density in such regions and thus potentially increases the probability for requested data to be cached by a cache server.

In addition, the gravity increases with the workload ω_n of a cache server. Thus, an overloaded cache server pulls the cache foci of neighboring cache servers closer to the position of its own cache focus. This increases the probability that future requests are forwarded to the neighbors of the overloaded cache server.

The application-defined parameter α determines how much each of the two aspects (workload or data density) is taken into account. For high α values the data skew is favored and vice-versa, the workload importance increases for low α values.

Finally, with the application-defined parameter β, the sensitivity of the particle-spring system can be tuned. It determines the responsiveness of

the spring contraction in the face of load. A high value makes the system very sensible to varying load, whereas a low value dampens the effects of load on the particle spring system.

The application-defined parameters α and β influence the value of the gravity measure and thus the behavior of the whole adaptation process. With these parameters, the application can tune the adaptation process of the distributed spatial cache towards specific adaption requirements. In Section 5, we provide tools that help to determine suitable values for these parameters.

Also, note that a cache server's gravity is always greater than one. This enforces a basic tension on the underlying particle-spring system. The basic tension is essential to ensure the physical stability of the particle-spring system, as loose springs could cause uncontrollable movements of the particles. With a static tension in the springs, it is possible to construct physical structures which stabilize the underlying particle-spring system. With this in mind, we intentionally designed the overlay topology of the distributed spatial cache in a way so that the underlying spring-particle system's physical stability is ensured. We present our approach in the following section.

4.2.1.2 Engineering a Particle-Spring-based Overlay Structure

In construction engineering, physical tension and the use of elastic materials are very common means to sustain stability while introducing flexibility at the same time. However, these constructions typically require static fixations at certain locations so that tension can build up in the elastic parts of the construction. Suspension bridges are examples for such flexible but physically stable construction designs. Suspension bridges are typically attached to a very stable anchorage which must support massive tension stress. With the same intention a construction engineer includes these static fixations into his/her construction design, we include such fixations in the design of the logical link structure of our distributed spatial cache.

For this reason, we introduce the concept of an *anchor* which is a fixed point in space. To construct physically stable particle-spring systems, these anchors must be inserted at dedicated locations. Figure 4.8 depicts an ex-

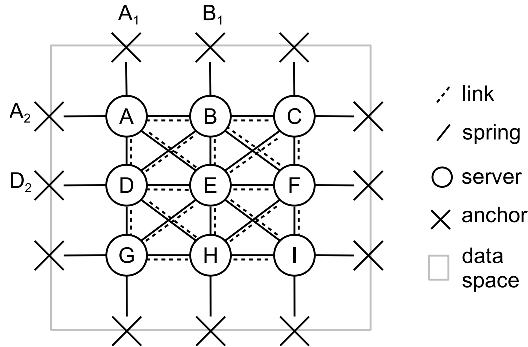


Figure 4.8: Particle topology

ample for a stable particle spring system design. In this design, a region of free-movable particles is supported by a structure of surrounding anchor points. To provide further stability, the particles are connected by springs in a horizontal, vertical and diagonal manner.

On top of the stabilized particle-spring system, we define an overlay topology of cache servers. For this purpose, each cache server keeps a *routing table* containing up-to-date information about the associated neighbors and the corresponding section of the particle spring system (see Table 4.1). These tables are periodically updated by piggybacking current values on heartbeat messages, which are used to detect server failures (see Section 4.2.1.5). A routing table contains entries which represent the basic items of the particle-spring system: free-movable particles and fixed anchors. The **id** attribute of the routing table corresponds to the network identifier of the associated cache server, such as its IP address. The **gravity** value indicates the current gravity and the **position** value contains the current position of the associated cache server's cache focus. Anchor entries are distinguished from entries of free-movable particles by a characteristic identifier. In contrast to free-movable particles, the **position** and **gravity** values of anchors stay fixed during the load-balancing process. The complete overlay topology and the state of the underlying particle spring system is

id	gravity	position
B	1.0	(3,4)
D	1.0	(2,3)
E	1.0	(3,3)
A_1	1.0	(2,5)
A_2	1.0	(1,4)

(a) Server A

id	gravity	position
A	1.0	(2,4)
B	1.0	(3,4)
C	1.0	(4,4)
D	1.0	(2,3)
F	1.0	(4,3)
G	1.0	(2,2)
H	1.0	(3,2)
I	1.0	(4,2)

(b) Server E

Table 4.1: Routing tables

represented by the routing tables of all cache servers. For instance, the routing table of cache server A in Table 4.1 states that A is connected to the three cache servers B, D and E and that A is fixed by the two anchors A_1 and A_2. In contrast, the routing table of cache server E does not contain any anchor entries, as it is solely surrounded by free-movable particles representing the cache foci of neighboring cache servers.

4.2.1.3 Implementing Load Balancing as a Self-governed Process

The cache servers periodically exchange the current position of their cache foci and the corresponding gravity values. To reduce the number of messages, the up-to-date information is piggy-backed on the heartbeat messages which are used to detect server failures (cf. Section 4.2.1.5). Every time a cache server receives such a heartbeat message, it updates the corresponding values in its routing table. This causes tension imbalances in the particle-spring system, as springs with a high stiffness pull with more force than soft springs. In a particle-spring system with tension imbalances, the movable particles snap into a position corresponding to the system's state of lowest overall energy. To find the low energy state, we base upon a simple iterative method which can be applied in a distributed setting. In the distributed approach, each cache server independently determines the

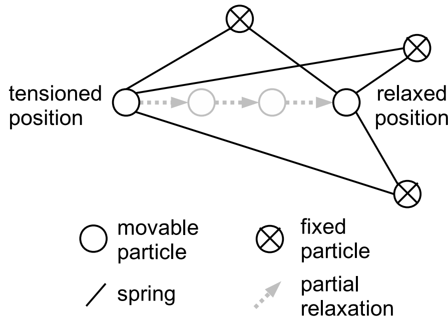


Figure 4.9: Iterative spring relaxation into low energy state.

positions of its own cache focus based on the current values of its routing table. The general idea is to move each particle a small distance into the direction of the cumulative force vector of attached springs during each iteration. As each iteration relaxes the involved springs, the amount of force in the particle-spring system is gradually decreased and the system eventually converges in a stable state of minimal overall energy, as depicted by Figure 4.9.

Algorithmic variants of the iterative spring relaxation have also been used in other contexts, e.g., make operator placement decisions in distributed stream processing systems [Pietzuch et al., 2006] or to compute network coordinates in a virtual latency space [Dabek et al., 2004]. Listing 4.1 shows our implementation of the iterative spring relaxation approach in pseudo code. For each neighbor n_i, the cache server computes the corresponding force vector of the attached spring. The length of the force vector is calculated by scaling the vector defining the spring's extension $\overrightarrow{nn_i}$ with the sum of the gravities $g_n + g_{n_i}$ of attached cache servers. The total force vector $\vec{F}$ is obtained by summing up all the separate forces. The resulting force vector $\vec{F}$ indicates the movement direction of the cache focus. During each iteration, the cache focus is moved a small amount into this movement direction. The length of each movement iteration is determined by the damping factor Δ. The algorithm terminates when the length of the

movement falls below a threshold τ which determines the precision of the cache focus calculation. The algorithm converges for values of $\Delta < 1$, as each iteration reduces the length of the resulting force vector $\vec{F}$.

```
1:  function CalculateCacheFocus
2:  input:
3:    n⃗: current cache focus position of cache server n
4:    g_n: current gravity value of cache server n
5:    {n⃗_1, n⃗_2, ...n⃗_k}: current cache foci of cache server n's neighbors
6:    {g_{n_1}, g_{n_2}, ...g_{n_k}}: current gravity values of cache server n's neighbors
7:  output:
8:    New cache focus position of cache server n
9:  loop
10:     F⃗ ← 0⃗
11:     for all n⃗_i ∈ {n⃗_1, n⃗_2, ...n⃗_k} do
12:         F⃗ ← F⃗ + nn_i⃗ · (g_n + g_{n_i})
13:     end for
14:     if |F⃗| < τ then
15:         return n⃗
16:     else
17:         n⃗ ← n⃗ + F⃗ · Δ
18: end loop
```

Listing 4.1: Iterative spring relaxation algorithm

4.2.1.4 Scaling

In the previous section, we covered load imbalances between cache servers. However, when the overall capacity of the distributed spatial cache is insufficient to process incoming load, resources have to be added or removed from the system. In the following, we describe the necessary steps to add and remove cache servers to the cache overlay. Subsequently, we propose a fully automated scaling mechanism.

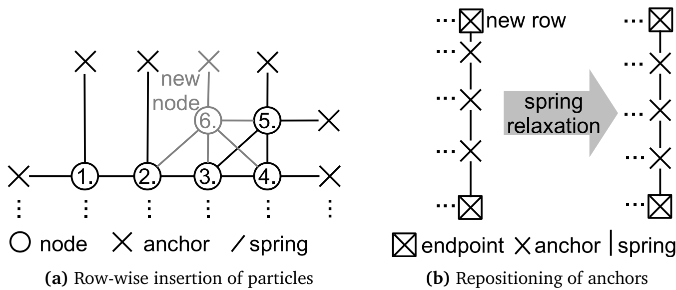
Adding Servers: When a cache server is added, a join request is send to an arbitrary server which is already part of the overlay. Once a join request

arrives at a cache server, an adequate position for the new cache server has to be found. For this process, the physical stability has to be considered. If cache servers joined without any order, the particle-spring system would deform and become unstable. As the edge particles are connected to at least one fixed anchor, their physical stability is much higher than the one of the free-movable particles in the center. Therefore, the edge regions of the particle-spring system offer acceptable positions for adding new particles.

Consequently, join requests are routed to a border by following the routing table's entries into the respective direction. Once they arrive at the border, the corresponding particles are added in a row-wise manner, as indicated by Figure 4.10a. The correct insertion position in the actual row of particles can be inferred from the ordered structure of the particle-spring system. Details about the insertion process can be found in our paper [Lübbe et al., 2012].

The insertion of particles at the border zones of the particle-spring system increases the particle density in these regions. For this reason, new particle rows are pushed into the center in order to disperse the particles uniformly across the space. To move complete rows of particles, the anchors supporting the particle rows must be realigned. This can be achieved in a distributed fashion by using the principle of spring relaxation. The general idea is sketched in Figure 4.10b. All anchors at the edge of the particle-spring system are connected by springs. The endpoints of these anchor chains are fixed, while midway anchors can move freely. The springs between the anchors possess a constant stiffness so that their spring force only depends on the spring's extension. The cache servers at the border of the particle-spring system periodically update the positions of their anchors. In this process, the anchors can move freely along the border line. The chain-like particle-spring system of anchors then eventually converges into the low energy state which corresponds to equal distances between the anchors.

Removing Servers: As the edge regions of the particle-spring system are stabilized by fixed anchors, the removal of particles in these regions has little effect on the overall stability of the system. Therefore, particles are removed from the border of the system in the reverse order as they

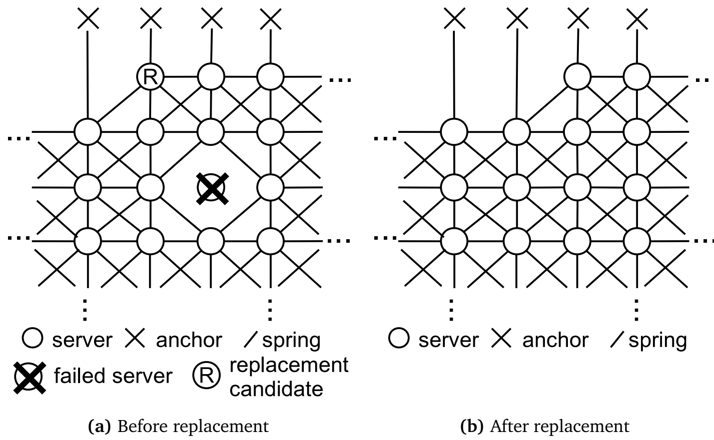


(a) Row-wise insertion of particles **(b)** Repositioning of anchors

Figure 4.10: Scale-out of the Distributed Spatial Cache

have been inserted. The cache server which corresponds to the last inserted particle can be inferred from the overlay structure, as described in our paper [Lübbe et al., 2012]. In consequence, any removal request arriving at an arbitrary cache server is forwarded to the cache server which has been inserted lastly. This server is detached from the overlay by updating the routing tables of neighboring servers accordingly.

Scaling: Implementing a self-governed scaling mechanism in a network of completely independent cache servers is not trivial, as the servers have to reach a consensus on the scaling decision, i.e., whether to add or remove a cache server. As existing consensus algorithms are very expensive in terms of communication costs, we do not recommend the self-governed approach as a practical solution. In a governed approach, a central master has to keep track of all the information relevant for the insertion process, i.e., the overlay structure, the order of insertion and so forth. To protect the system against failures of this master, its complete state has to be continuously mirrored to a back-up system. As the centrally stored state is huge in the governed approach, we do not recommend this approach for scaling either. Instead, we propose a supervised implementation which minimizes the amount of central state. The general idea is that a central manager component takes samples from some of the cache servers to obtain a global measure characterizing the overall utilization of the system, e.g. the av-



(a) Before replacement

(b) After replacement

Figure 4.11: Replacement process of failed cache servers.

erage request processing latency or average CPU utilization. Based on this measure, the manager makes a scaling decision and adds or removes servers from the overlay using the mechanisms previously described.

4.2.1.5 Maintenance

To detect failures, cache servers periodically send heartbeat messages to their neighbors. The absence of such a heartbeat message indicates a failure of a cache server. As soon as a failure is detected, a replacement process for the missing cache server is triggered. The general procedure of this replacement process is sketched in Figure 4.11. During the replacement the physical stability of the underlying particle-spring system has to be ensured. Therefore, the particles surrounding the failed cache server's position freeze to their current position. To achieve this, the cache servers neighboring the failed server stop to update their cache foci during the replacement process. This prevents the particle-spring system to collapse or deform when intermediate spring connections are missing. The missing cache server must

then replaced by another cache server of the distributed spatial cache. As the last inserted server might not have filled up its cache yet and therefore might not be fully operational at this time, it is likely that a reallocation of this server causes less impact than the reallocation of an already fully operational server. Therefore, we use the last inserted cache server at the border of the system as the replacement candidate. This replacement candidate can be inferred from the overlay structure of the distributed spatial cache. Further details about inference of the replacement candidate can be found in our paper [Lübbe et al., 2012].

4.2.1.6 Query Processing

Exploiting the geographically arranged link structure of our particle-spring model, we apply a greedy forwarding mechanism in order to route a user requests to the cache server possessing a cache focus close to the target region. In our basic query processing mechanism this server then processes the request. In addition to that, we also investigated several extensions to this basic query processing mechanisms. Our extensions base on the assumption that any direct neighbor of the server at which the forwarding terminates is also close – although not closest – to the target region and thus may also have cached relevant data. Therefore, it may be reasonable to also include the direct neighbors into the processing. In this way, the hit rate can be further increased or some local overload situation can be relieved. For reasons of conciseness, we refrain from a detailed description of our query processing extensions and refer the reader to [Lübbe et al., 2012] for more details.

4.2.1.7 Evaluation

In this section, we present the most significant parts of our evaluation we conducted in a previous work [Lübbe et al., 2012]. For further details, we refer the reader to that paper. In the following, we sketch the conducted experiments and discuss the most significant results of our evaluation.

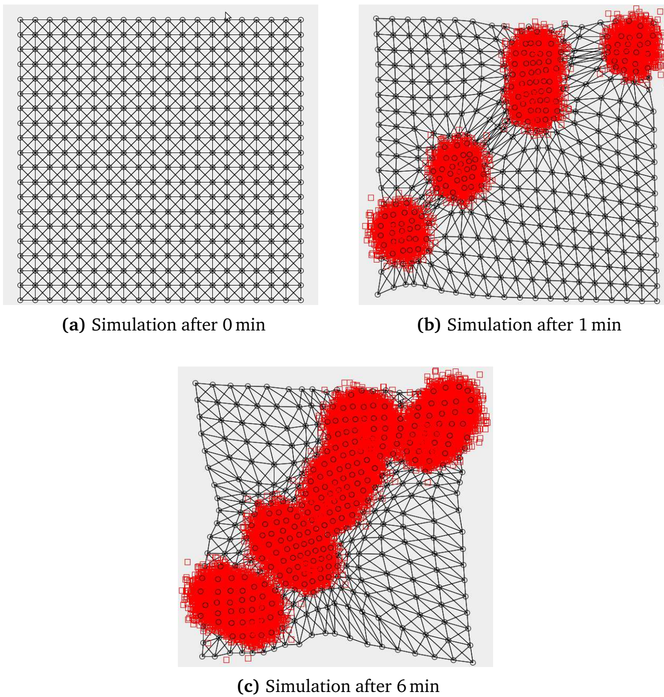
Experimental Setup

Our evaluation bases on PeerSim [Montresor and Jelasity, 2009], a framework for discrete event-driven simulation of peer-to-peer networks. With this framework, we created a particle-spring-based implementation for the cluster layer and included it into the simulation. We simulated a distributed spatial cache comprising a total number of 400 cache servers. Each cache server has the capacity to store 250 objects. The servers were initialized with empty caches forming a uniform particle topology.

The simulation uses real world data of Berlin and its outskirts extracted from OpenStreetMap [OpenStreetMap, 2011]. It comprises a total number of 23625 objects. The data set covers a region of 40 km × 48 km in which the data is not spread uniformly, but rather resembles the density of the city's development.

On top of this data, we simulated the data access behavior of clients which request data surrounding their current position using 500 m × 500 m range queries. For this reason, the simulation bases on trajectories of mobile objects which follow streets on the city map changing their speed in between 30 km/h and 60 km/h. For each trajectory, we generated range queries in equidistant time intervals of 20 ms. The target regions of the range queries are scattered around the current trajectory position according to a Gaussian distribution with a deviation of 1000 m. The simulation continues for 10 min which sums up to a number of 30000 queries per trajectory. The access behavior of clients resembles hot spots which move in space and time. This could be possibly induced by clients which request data surrounding their current position while stuck in a traffic jam. We simulated 5 of such moving hot spots simultaneously, resulting in a total number of 150000 client requests.

During simulation, we executed the iterative spring relaxation algorithm (see Listing 4.1) using $\Delta = 0.005$ and $\tau = 2$ to adapt towards the shifting load. Figure 4.12 visualizes the distributed spatial cache at different simulation times. Red squares resemble the range queries, black circles visualize the cache server's cache foci and black lines depict the springs of the particle-spring system. We omitted the anchors for reasons of visibility.



(a) Simulation after 0 min

(b) Simulation after 1 min

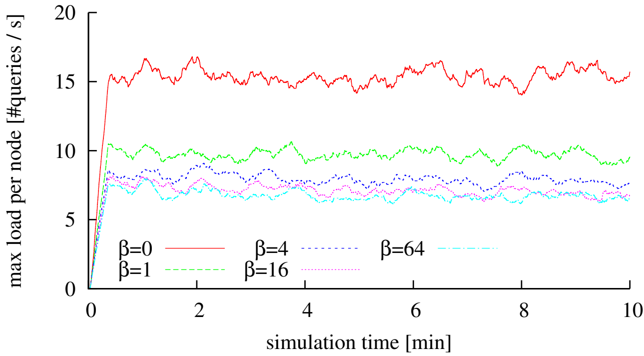
(c) Simulation after 6 min

Figure 4.12: Visualization of the distributed spatial cache

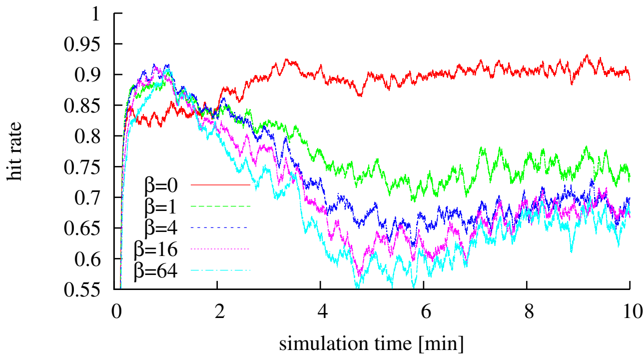
During the simulation, we investigated the hit-rate and the maximum load per cache server. We refer the reader to Section 4.1.1.6 for an exact definition of these measures. In the following, we discuss the results of the experiments.

Main Results

First, we examine the impact of the gravity scale parameter β (cf. Section 4.2.1.1) on the maximum workload of all cache servers in the distributed spatial cache. During this experiment, only the workload con-



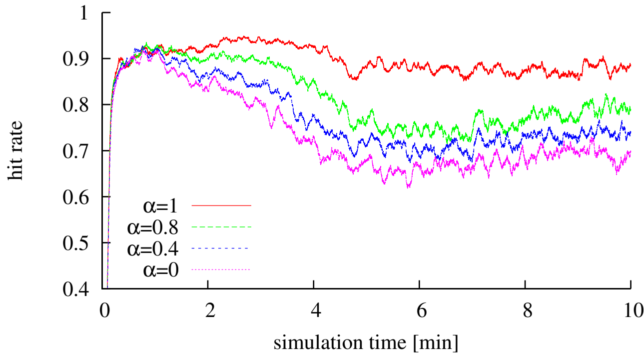
(a) Maximum workload



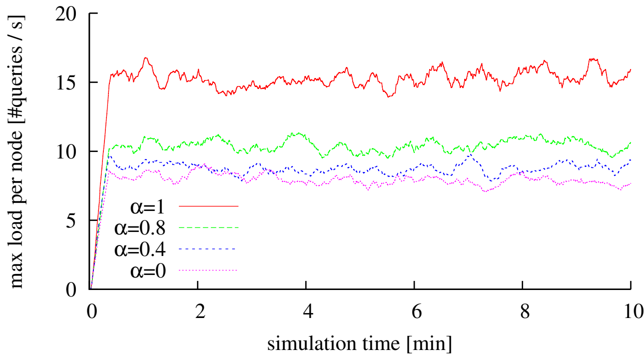
(b) Hit rate

Figure 4.13: Impacts of gravity scale parameter β

tributes to the particle gravities of the cache servers (i.e., the load weight parameter is set to $\alpha = 0$). Figure 4.13 visualizes the maximum workload during the simulation for different values of the gravity scale parameter β. For $\beta = 0$, only static tension affects the underlying particle-spring system, i.e., all servers stay fixed at their initial position and do not adapt to varying



(a) Hit rate



(b) Maximum workload

Figure 4.14: Impacts of the load weight parameter α

load patterns. It can be observed that higher values of β clearly reduce the maximum workload and thus improve the workload distribution among the cache servers. The gravity scale parameter can also have a positive effect on the hit rate, as it can be observed in Figure 4.13b. In the first two minutes of simulated time, higher values of β outperform the static particle-spring

system. With the higher responsiveness of the particle spring system, the springs contract quickly and cache servers form clusters in the centers of the data access hot spots. Thus, the cache capacity in the centers of the hot spots increases. This reduces cache misses and explains the higher hit rate during that time. However in the course of time, the hit rate degrades and the static system shows better values. As the cache servers change the position of their cache foci in order to adapt to the moving hot spots, their cache content becomes invalid. This causes a higher number of cache misses and degrades the hit rate.

In a further experiment, we examine the impacts of the load weight α on the hit rate. In this experiment, the gravity scale parameter was set to $\beta = 4$. Figure 4.14a depicts the hit rate during the simulation. A data density favored setting (high α values) induces higher hit rates compared to a workload centered approach (low α values). The load scale parameter has a contrary effect on the workload distribution, as it can be observed in 4.14b. In this case, low values of α reduce the maximum workload in the distributed spatial cache and high values increase it.

The experiments show that optimizing the workload distribution can have a contrary effect on the hit rate and vice versa. In general, this implies that it is impossible to find a solution in which both the hit-rate and the workload distribution are optimal at the same time. However, with the application-defined parameters α and β an application is able to define a compromise between the two contrary optimization goals that suits specific adaptation requirements.

4.2.1.8 Discussion

Our evaluation confirms the principal feasibility of the particle-spring-based cluster layer. Exploiting spring contraction, we could adapt the distributed spatial cache to current data access patterns and thus improve the workload distribution between the cache servers. Furthermore, by considering the data skew, we could achieve cache hit rates over 90 %.

However, the required resources to achieve such high hit rates are comparatively high: In our experiment, we required 400 cache servers each of

which having a capacity to store 250 objects. In total this approximately sums up to four times the amount of the data set's size. The reason for this effusive resource consumption can be observed in Figure 4.12. In our example, the rectilinear construction layout of the particle-spring system stretches over a vast area while the actual data access is quite selective, i.e., it is limited to certain areas around the moving hot spots. Even though many cache servers adapt their cache focus to the data access hot spots, still some cache focus points are allocated to geographic regions which are hardly requested. For this reason, many resources are bound without being used at all. This suboptimal allocation of cache foci is caused by the structural properties of the particle-spring construction: The fixed anchors at the border of the particle-spring system hold on to the static tension between the cache focus points. This prevents cache focus points which are far away from the hot spot areas to be pulled into these areas.

In conclusion, the integral properties of the physical model seem to counteract optimal resource allocation for selective data access patterns. For this reason, we propose an alternative approach particularly suited for selective data access patterns in the following section.

4.2.2 A Delaunay-based Implementation

The previous section presents an implementation of the cluster layer which bases upon a particle-spring system. Our evaluation of this approach confirms its principal viability in a realistic application scenario. For selective data access patterns, however, we identified inherent characteristics of the particle-spring system's physical model that prevent an effective resource utilization. For this reason, we subsequently present an alternative model as a basis for the cluster layer which is particularly suited for selective data access. Thereby, we summarize the main contributions of two preceding publications [Lübbe and Mitschang, 2013b, Lübbe and Mitschang, 2013a] to which we refer the reader for further details.

Our approach is based on the mathematical model of Delaunay triangulations. Informally, a triangulation of a discrete point set is an arrangement of non-overlapping triangles in which the points constitute the triangle ver-

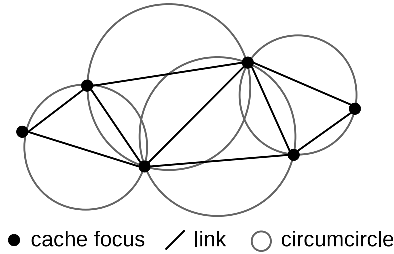


Figure 4.15: A Delaunay-based link topology with circumcircles

tices. In our context, the triangle vertices represent the cache foci and the triangle edges constitute the links between the corresponding cache servers. A triangulation is called *Delaunay triangulation* if it adheres to the Delaunay property, i.e., no triangle vertex is inside the circumcircle of any other triangle of the triangulation (a formal definition can be found in [de Berg et al., 2000]). Geometrically, the goal of the Delaunay property is to prevent slim triangles in the triangulation. This creates a linked topology in which distance-based forwarding can be applied. In a distance-based forwarding mechanism, a client request is iteratively forwarded to the cache server owning the closest cache focus to the target region until no closer cache focus is found. In a Delaunay-based link topology of processing nodes, it can be guaranteed that the forwarding always terminates at the node owning the closest vertex to the target coordinate (a formal proof can be found in [Bose et al., 1999]). This is an essential property to ensure that client requests are processed by the most suitable cache server, i.e., the cache server which is most likely to have cached the requested data. Figure 4.15 depicts an example for a link topology which adheres to the Delaunay property as none of the cache focus points is located inside the circumcircle of a triangle.

In contrast to the particle-spring-based link topology, the Delaunay-based topology is not restricted by the low-level physical properties of the particle spring system. This increases the flexibility during construction and maintenance of the link topology. For instance, it is very easy to add or remove

servers in a Delaunay-based topology. In a particle-spring-based topology, quite complex measures have to be taken to ensure the physical stability of the system. Moreover, the Delaunay-based topology does not need to be stabilized by fixed anchors. This allows the reproduction of arbitrary geometric shapes in order to match the current data access patterns. Thus, even a very selective data access pattern can be facilitated.

In the following, Section 4.2.2.1 describes the basic mechanisms required to construct and maintain a Delaunay-based link topology. Based on these principle mechanisms, we propose a load balancing mechanism in Section 4.2.2.2. Subsequently, we devise a load model suitable for our target application field in Section 4.2.2.3. Finally, Section 4.2.2.4 introduces a scaling approach.

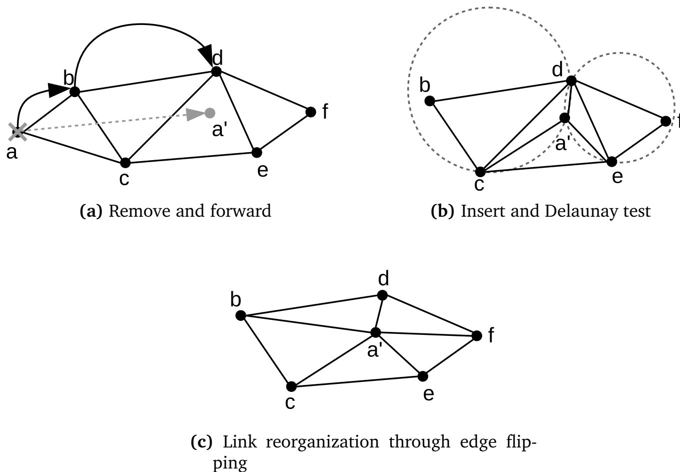
4.2.2.1 Constructing, Maintaining and Adapting a Delaunay-based Link Topology

To construct and maintain Delaunay-based link topologies a distributed algorithm is proposed in [Lee and Lam, 2008]. The algorithm bases upon two distributed mechanisms: one mechanism to add new nodes to a Delaunay-based network of processing nodes and one mechanism to remove existing nodes. To add a new node, a new triangulation can be obtained from the triangular link topology of processing nodes which are already part of the network. For instance, if the new node's vertex is located inside a triangle surrounded by the vertices of other processing nodes, the new node's vertex is connected to all vertices of the encircling triangle. If the thus obtained triangles do not comply with the Delaunay property, their edges are flipped according to the edge flip algorithm [Lawson, 1977] until all triangles adhere to the Delaunay property. For node removal, the adjacent triangles of the leaving node's vertex are removed from the triangulation. If this leads to unconnected vertices, additional edges are inserted to restore the triangulation. Then, the triangulation is checked for Delaunay coherence and possibly corrected using the edge flip procedure. The two mechanisms can be straightforwardly adapted for the purpose of adding/removing cache servers to/from our distributed spatial cache.

In addition to that, an additional mechanism is required to move a cache focus of an existing cache server to another position which is necessary for load balancing (see Section 4.2.2.2). This can be accomplished by building upon the existing join and remove mechanisms. In the following, we briefly sketch the basic procedure to move an existing cache server's cache focus:

1. **Remove:** The cache server's cache focus is removed from the triangulation using the existing removal mechanism. The mechanism then re-establishes a Delaunay-coherent link topology.
2. **Forward:** Afterwards, distance-based forwarding to the new position of the cache focus is used to determine the closest cache server to the new insert position.
3. **Join:** The previously identified cache server inserts the new cache server into the distributed spatial cache using the existing join mechanism. The outcome of this procedure is a Delaunay-coherent topology again.

Figure 4.16 depicts an example for a cache focus movement in a Delaunay-based topology. In the example, the cache focus a is to be moved to position a' into the center of triangle (c, d, e). In Figure 4.16a, the cache focus a is removed. Then distance-based forwarding (visualized as black arrows) identifies the cache server possessing the closest cache focus to the destination position a' which is the cache server owning the cache focus d in our case. This cache server updates the link topology in cooperation with the other cache servers of the destination triangle (c, e, d) which are the cache servers owning c and e in the case of the example. For this purpose, the destination triangle is split into three new triangles (c, a', d), (e, d, a') and (c, e, a'), as depicted in Figure 4.16b. Afterwards, the new triangulation is checked for Delaunay coherence. In our example, the new triangles (c, a', d) and (e, d, a') both violate the Delaunay property, as b and f are contained by the corresponding circumcircle. Then the edges (c, d) and (d, e) are flipped to obtain a Delaunay coherent triangulation according to Figure 4.16c.



(a) Remove and forward

(b) Insert and Delaunay test

(c) Link reorganization through edge flipping

Figure 4.16: Move a node's cache focus position

4.2.2.2 Load Balancing

The previous section describes the basic mechanisms to establish, maintain and adapt a Delaunay-based link topology. In this section, we present a load balancing mechanism which continuously adapts the topology towards given data access patterns. A detailed description of this approach can also be found in [Lübbe and Mitschang, 2013a].

In an abstract understanding, we consider the cache foci of the cache servers as the available resources. By situating a certain cache focus in a specific location in space, the distributed spatial cache can make use of the corresponding cache server's cache memory and processing capabilities for processing requests referencing the area surrounding the location of the cache focus. Thus, the distribution of cache foci in space determines the actual amount of resources allocated for certain areas. In practice, the actual resource distribution seldomly matches the current demand, as clients can change their data access behaviour. Therefore, the load-balancing process

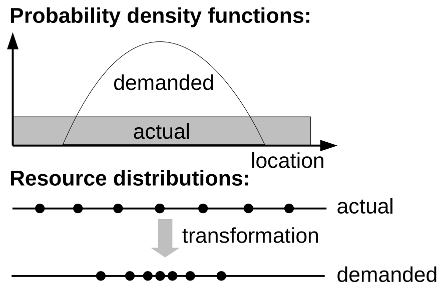


Figure 4.17: From the actual to the demanded distribution of resources

must continuously adapt the actual resource distribution to the demanded resource distribution.

Mathematically, we can describe a distribution of resources in space as a two dimensional probability density function (pdf) $\rho : \mathbb{R}^2 \rightarrow \mathbb{R}$. The function returns a value which determines the likelihood of a resource being allocated to a given location in the geographic space. Thus, if the function returns higher values for locations in certain areas, the density of resources should increase for that areas. We use the model of probability density functions to characterize both the actual resource distribution and the demanded resource distribution. The objective of the load balancing process is to transform the actual resource distribution into a distribution which matches the current demand.

Figure 4.17 shows an example for such a pdf-based load-balancing process in a one-dimensional simplification. It may resemble a scenario in which mobile clients move along a certain street and request data at their current street position. In the example, the actual distribution of resources is uniform so that the distributed spatial cache offers equal processing capabilities at any point along the street. This resource distribution is adequate when the mobile clients are equally distributed along the street. Now, imagine a traffic jam at a certain street position. In this case, the pdf of the demanded resource distribution may resemble a Gaussian distribution cen-

tered around the location of the traffic jam. The goal of the load balancing process is to condense the cache focus positions in the area of the traffic jam so that the available resources increase in that area. Our one-dimensional example can be easily generalized to two dimensions.

To transform the actual resource distribution into the demanded resource distribution, we base upon the Random MacQueen's method which is derived from the traditional K-Means algorithm [Macqueen, 1967]. In another context, the algorithm is used to obtain centroidal Voronoi diagrams [Du and Gunzburger, 2002]. While the traditional K-Means algorithm finds cluster centers in point set distributions, the Random MacQueen's method transforms a set of input points into a set of output points matching a given distribution function.

procedure CALCULATENEWCACHEFOCUSPOSITIONS
input:
 ρ: distribution function expressing the current resource demand
 $\{\vec{n_1}, \vec{n_2}, ...\vec{n_k}\}$: actual cache focus positions of cache servers
output:
 $\{\vec{n_1}, \vec{n_2}, ...\vec{n_k}\}$: new cache focus positions of cache servers
begin
 Initialize indices $i_j \leftarrow 1, \forall i_j \in \{i_1, i_2, ..., i_k\}$
 repeat
  $\vec{r} \leftarrow \text{rnd}(\rho)$, where rnd generates a random point according to ρ
  $\vec{n_j} \leftarrow \frac{i_j * \vec{n_j} + r}{i_j + 1}$, where $\vec{n_j}$ is the closest cache focus to r
  $i_j \leftarrow i_j + 1$
 until Average distance of the last λ movements is below threshold τ
 return $\{\vec{n_1}, \vec{n_2}, ...\vec{n_k}\}$
end

Listing 4.2: Algorithm for a ρ-distributed topology of cache foci

We adapted the Random MacQueen's method to transform the actual distribution of cache focus points into a distribution matching the current demand given by the distribution function ρ. Listing 4.2 depicts the algorithm. In the interpretation of k-means, the points resemble the centers of k clusters. In each iteration, the algorithm generates a random point r according

to ρ, finds the closest cluster center n_j and updates it. The algorithm terminates when the average distance of the last λ movements is lower than the threshold τ. The application defined parameters λ and τ determine the accuracy of the output, i.e., how much the calculated distribution matches the demand, given by the probability density function ρ.

The algorithm can be executed independently by each cache server using the cache focus points of the cache server and its immediate neighbors as input. In such a distributed execution, the input point set resembles just a subset of the cache focus points of all cache servers. If the algorithm is executed in a distributed manner, the mechanisms described in Section 4.2.2.1 can be used to update the link topology for all changed cache focus positions. If the algorithm is executed centrally in a supervised or governed architecture (cf. Figure 2.4), a standard triangulation algorithm, such as the Edge-Flip algorithm [Lawson, 1977] suffices.

The approach described above is a generic way to adapt a Delaunay-based topology to changed demand expressed by an arbitrary probability density function. In the following section, we design specific probability density functions which are particularly suited in the context of scenarios with selective data access.

4.2.2.3 Modeling Spatial Load

The aforementioned load balancing approach is general, as arbitrary probability density functions can be included to express application specific load balancing requirements. In this section, we design probability density functions that can express the occurring data access patterns of our target application field. For this purpose, we classify the probability density functions into the categories of *static*, *dynamic* and *hybrid*. This classification bases upon previous work [Lübbe and Mitschang, 2013b]. Each category covers specific requirements of our application field. In the following, we detail on the categories and provide conceivable examples for probability density functions of each category.

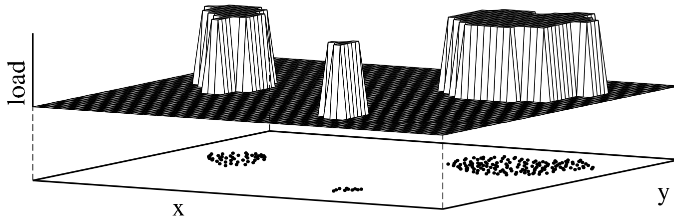


Figure 4.18: Geographic distribution ρ_G

Static Probability Density Functions

We denote probability density functions which can be formed before runtime as *static*. They can express prior knowledge on data skew, statistical analysis of previous workloads, heuristic assumptions about future workloads or even application specific resource allocation needs. A static probability density function enables to prepare for anticipated situations before the actual load strikes the system. Thus, they can be a major indication for allocating resources in an adequate manner.

To design suitable probability density functions of this class, we consider a use case of a coastal region containing small insular regions and a huge offshore portion. For an electronic bicycle and hiking guide the offshore regions are probably of little interest, as bikers and hikers typically request onshore data. In the context of this example, we can model a function, which excludes all offshore areas. Given the extent of the insular parts in geometry G, we define the distribution function ρ_G which returns 1 for insular regions and 0 otherwise:

$$\rho_G(x,y) = \begin{cases} 1, & \text{iff } (x,y) \in G \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

Figure 4.18 depicts a resource allocation according to ρ_G. As observable, the function leads to a uniform resource distribution of cache foci within land parts and a total cutout of sea areas. Despite its simplicity, this function can be quite beneficial for reaching an adequate resource allocation, as

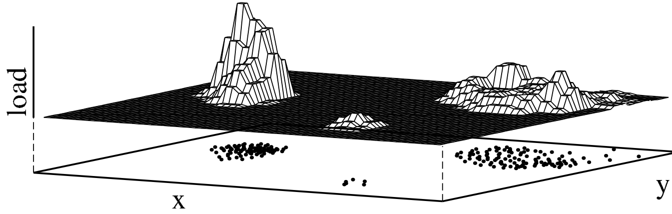


Figure 4.19: Data skew ρ_N

approximately 70 % of the world is covered with water. In comparison, a uniform distribution function would dedicate most of the resources to sea areas which are not required by the application.

Beyond that, we can model data skew as a distribution function. Intuitively, such a data skew function should provide a measure for the quantity of expected data at a certain geographic position. This can serve as an indication of how much cache memory should be allocated to certain regions in order to store all relevant data. Given a discretization of the geographic space in a uniform grid of parcels, we can define the distribution function ρ_N expressing the geographic data skew. For given coordinates (x, y), ρ_N returns the number of objects o in the corresponding parcel P:

$$\rho_N(x, y) = |\{o | o \in P, (x, y) \in P\}| \tag{4.2}$$

In Figure 4.19, the function values of ρ_N are visualized in the upper part and the corresponding resource allocation on the plane at the bottom part of the figure. As it can be observed, the data density is higher within the left island which leads to a denser clustering of cache focus points in this area.

Dynamic Probability Density Functions

We denote probability density functions which can be only manifested at run-time as *dynamic probability density functions*. Such probability density functions can be used to characterize the current resource demand of the ap-

plication at any point of time. To obtain a dynamic probability density function, the current system state has to be included. This is typically achieved by continuous observation of the relevant system components. From that characteristic data, the return value of the function at a certain geographic position can be extrapolated.

We design such a dynamic probability density function for our target application field. Our exemplary function provides a measure for the request frequency of geographic areas. Thus, for positions in highly requested areas, the function should return high values and vice versa, low values for unpopular regions. To achieve this behavior, we include the current workload of the cache servers into the computation of the function values. For this purpose, we use the cache focus positions to situate the current workload of cache servers in the geographic space. With this input, we can define the distribution function ρ_D for a given set of k cache servers:

$$\rho_D(x,y) = \sum_1^k \frac{\omega_j}{1 + d(p_j,(x,y))} \tag{4.3}$$

Thereby, ω_j denotes the current position of cache server j's cache focus and ω_j its current load (incoming requests per second). The function d is the Euclidean distance. ρ_D returns the average load of the cache servers, whereas a cache server's load ω_j is weighted by the reciprocal distance to the given coordinates (x,y). Thus, cache servers which possess a cache focus near the requested coordinates contribute more to the result value than cache servers with a far away cache focus.

Opposed to the static probability density functions, the dynamic probability density function ρ_D changes during the runtime. The changes are reflected by varying workload values ω_j of the cache servers. These values must be repeatedly collected during the runtime. The above definition of ρ_D resembles the instant in time when the cache server workloads ω_j have been collected.

As depicted by the top part of Figure 4.20, the function outlines peaks and drops, where a peak approximates the current workload of a cache server at the geographic location of its cache focus and the drops constitute

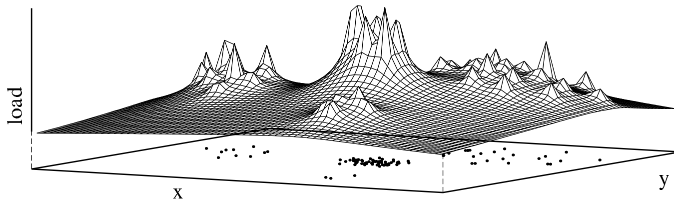


Figure 4.20: Dynamic load distribution ρ_D

a degradation dependent on the distance from surrounding workload peaks. The high peak cluster in the middle of the figure resembles an unusual raise of interest in an area which is usally of little concern. The high workload in this area could be possibly induced by an unforeseen event, such as a ship wreckage, oil spill or any other critical event which suddenly raises disproportionately high interest in the area around the event's location. As observable in the bottom part of Figure 4.20, a straightforward resource allocation using this dynamic probability density function, leads to a high quantity of cache focus points clustered in the center. With the background knowledge that only marginal data exists in this area, we can observe a clear oversupply of cache focus points in that area. For this reason, it can be necessary to include also static probability density functions into the computation. We do this in the following section.

Hybrid Probability Density Functions

As already indicated previously, load balancing which bases solely on dynamic probability density functions can lead to inadequate resource allocations. Therefore, it is necessary to incorporate static probability density functions to further improve the dynamic load balancing process. We denote probability density functions that combine static and dynamic probability functions as *hybrid*.

With the previously given probability functions we can form such a hybrid probability density function for our target application field. Thus, we define the probability density function ρ_H:

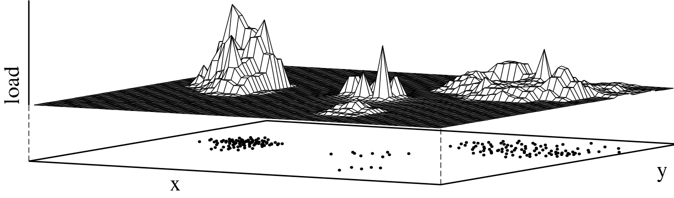


Figure 4.21: Hybrid load distribution ρ_H

$$\rho_H(x,y) = \delta * \rho_S(x,y) + (1-\delta) * \rho_D(x,y) \tag{4.4}$$

In the previous equation, ρ_S denotes a static probability density function such as our examples ρ_G or ρ_N or possibly a combination of them. δ denotes an application-defined weight parameter which allows fine grained tuning between the two functions. Using ρ_H, the application can relate in other respects unrelated probability density functions to achieve specific resource allocation demands.

As it can be observed in Figure 4.21, the hybrid probability density function makes sense in our application scenario. In this case, we incorporated the data skew ρ_N and the dynamic probability density function ρ_D in equal shares. The corresponding resource allocation on the bottom plane of the figure prefers workload in regions with high data density. This leads to a more adequate distribution of cache foci, as it does not overcompensate workload hotspots where only marginal data exists.

4.2.2.4 Scaling

With the mechanisms introduced in Section 4.2.2.1 nodes can be added and removed at any position. In contrast, the particle-spring approach only allows to add new cache server at the border region of the particle-spring topology. The flexibility of the Delaunay-based topology is advantageous in the context of scaling, as it enables to add resources where they are needed.

This enables the system to react quickly to the occurent overload situations of real-world scenarios.

In a student project in the context of this work, a local scaling mechanism was implemented [Gessler et al., 2014]. The general idea of this mechanism is that each cache server locally measures its workload (e.g., received queries per second). If the workload measure exceeds a given threshold, the cache server communicates with its neighbors to determine whether adding a new cache server is reasonable. If so, a new cache server is added at a position which is most beneficial in order to relieve the local overload situation.

4.2.2.5 Evaluation of the Adequateness of Resource Allocation

In this section, we present the most significant parts of our extensive evaluation [Lübbe and Mitschang, 2013a]. We refer the reader to this paper for further details. The evaluation focuses on examining the adequateness of resource allocation of the particle-spring and the Delaunay-based approaches. In the following, we define the investigated measures, briefly describe the conducted experiments and finally discuss the most significant results of our evaluation.

Investigated Measures

To determine the quality of resource allocation, we start with characterizing the adequateness of our caching strategy in respect to the given workload. We use cache hit rate to measure this aspect. We define the cache hit-rate of a given query at time t as the number of objects retrieved from the cache n_c divided by the number of objects satisfying the query n_q. If no object satisfies the request (i.e., $n_q = 0$), the ratio of n_c and n_q is undefined. As cache servers internally keep track of empty regions, they can often avoid to send a request to the data back-end in this case. Thus, we define the hit rate in the empty result set case as 1 if the back-end was not contacted and 0 otherwise:

$$hit_rate(t) = \begin{cases} \frac{n_c}{n_q} & \text{iff } n_q > 0 \\ 0 & \text{iff } n_q = 0 \wedge \text{back-end was contacted} \\ 1 & \text{iff } n_q = 0 \wedge \text{back-end was not contacted.} \end{cases} \quad (4.5)$$

Furthermore, we investigate the distribution of workload among the cache servers. To examine this aspect, we measure the deviation of all cache servers from the average workload in the system. If avg_t is the average workload of all n cache servers at a time t in queries per second and w_{tj} is the workload of cache server j at time t, then the average deviation $\overline{dev}$ can be defined as:

$$\overline{dev}(t) = \frac{\sum_{j=1}^{n} \omega_{tj} - avg_t}{n} \quad (4.6)$$

From that we can derive the normalized deviation dev as follows:

$$dev(t) = \frac{1}{1 + \overline{dev}(t)} \quad (4.7)$$

The normalized deviation dev constitutes our measure for the optimal workload distribution.

Conducted Experiment

To evaluate our approaches, we used PeerSim, a discrete event-driven simulation framework for peer-to-peer networks [Montresor and Jelasity, 2009]. We simulated a distributed spatial cache comprising a total number of 100 cache servers which were initialized with empty caches. Each data cache of a cache server has a capacity of 300 objects; i.e., the overall capacity of the distributed spatial cache sums up to 30 000 objects. In accordance with the use case presented in Section 4.2.2.3, the simulation uses real world data of the Italian island Giglio and its surrounding area extracted from OpenStreetMap [OpenStreetMap, 2011]. The data set includes a total number of 36 777 geographic objects, i.e., approximately 80 % of these objects fit into the cache. The data set covers a region of approximately 32 x 35 km

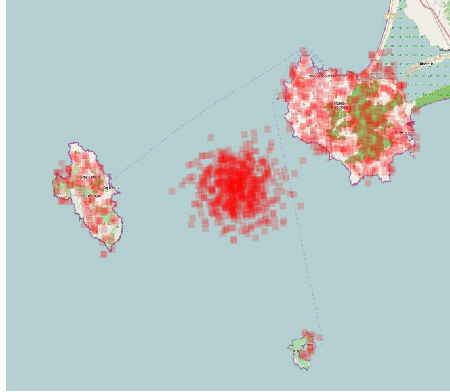
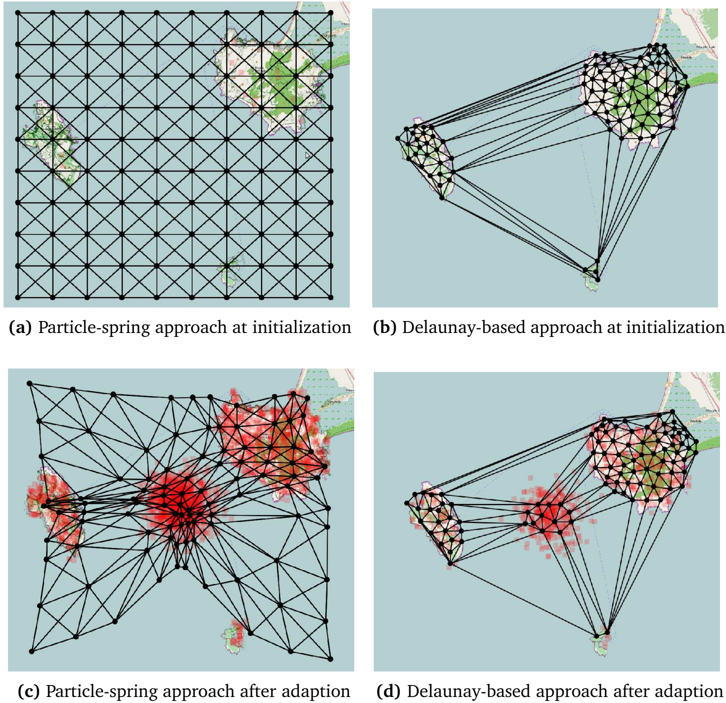


Figure 4.22: Data access pattern during simulation

in which data is not distributed uniformly, but rather resembles the density of geographic features in the onshore parts of the insular region. On top of this data, we simulated a workload of 500 x 500 m range queries at a rate of approximately 83 queries per second. I.e., every 12 ms a request was sent to a randomly chosen cache server in the simulated cache server cluster. We simulated a total number of 15 000 requests. The target regions of 10 000 of these requests were distributed uniformly across the land parts of the insular region. 5 000 requests simulated a data access hotspot in the sea area according to a Gaussian distribution with a standard deviation of 15 000 m. Figure 4.22 visualizes the range queries as red squares in the insular region.

In this evaluation, we compare the particle-spring based approach with the Dalaunay-based approach. In the following, we describe the relevant simulation settings for both approaches.

The particle-spring-based topology was initialized by distributing the cache foci uniformly across the data area in rows of 10 cache foci in each dimension (cf. Figure 4.23a). Using an application-defined parameter α (see Section 4.2.1.1), the load-balancing target of the particle-spring model can be tuned towards a metric characterizing the current workload or a metric



(a) Particle-spring approach at initialization **(b)** Delaunay-based approach at initialization

(c) Particle-spring approach after adaption **(d)** Delaunay-based approach after adaption

Figure 4.23: Illustration of the adaption process

approximating the data density. The metric expressing the data density of the cache content is defined as the number of geographic objects cached by a single cache server divided by the area the complete cache content covers. In most of the cases, this provides a good indication on how much cache memory should be allocated for certain areas. However, this measure tends to be imprecise for data sets including vast empty regions. This leads to non-optimal load balancing decisions in our example scenario. For that reason, we conducted all our experiments with a workload-only setting ($\alpha = 0$). The stiffness of the particle-spring system can be influenced by an

additional application-defined parameter (β). For each of our experiments, we empirically determined the most advantageous setting for this parameter and compared the best results with the results of our Delaunay-based approach.

The Delaunay-based approach was initialized using our distribution function ρ_G, i.e., the cache foci were distributed uniformly across the land parts of the insular area, as illustrated in Figure 4.23b. Thus, this initial resource allocation matches the larger part of the workload distribution. This experiment setup reflects an often occurring situation in practice in which some parts of the expected workload can be anticipated before run-time. During the simulation, our dynamic load balancing process (cf. Listing 4.2) is used to adapt to the unforeseen part of the workload (i.e., the request hotspot in the sea area). The dynamic load balancing process uses ρ_H which incorporates equal shares of ρ_G and ρ_D (i.e. $\delta = 0.5$). The accuracy parameters of the load balancing process were set to $\lambda = 20$ and $\tau = 50\,\text{m}$. The cache servers executed the load balancing process every 5 s of simulated time.

Results

Figure 4.24 depicts the values of the investigated measures. As it can be observed in Figure 4.24a, the Delaunay-based approach achieves higher hit-rates than the particle-spring approach. The main reason for the poor performance of the particle-spring approach are the physical constraints of the model which become quite evident in the insular application scenario. The hotspot in the sea area in the center of Figure 4.23c attracts so many cache focus points that the rest of the accessed areas are under-supplied with resources. The limited capacity of 300 objects per cache server in this experiment aggravates the effect. This experiment exposes the limits of the particle-spring approach in matters of selective data access patterns. In contrast, the Delaunay-based approach generally achieves a hit-rate around 50 to 60 %. With higher values for δ (i.e., favoring the static distribution ρ_G), the hit-rate increases, while lower values for δ (favoring the dynamic workload distribution ρ_D) slightly decrease the hit-rate. This is because ρ_G preserves many cache focus points for land parts as illustrated in Figure 4.23d. Contrary results can be observed for the dev measure in Figure 4.24b where low values of δ lead to a better distribution of workload among the cache

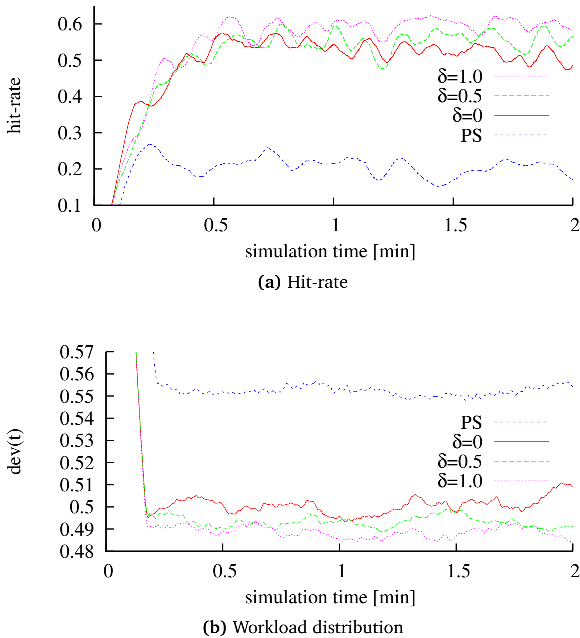


Figure 4.24: Results for the particle-spring apprach (PS) and the Delaunay approach with a static distribution ($\delta = 1$), a dynamic distribution ($\delta = 0$) and a hybrid distribution ($\delta = 0.5$)

servers. In this case, ρ_D ensures that the cache foci are drawn together in regions where the workload is high. In comparison to the particle-spring model, the Delaunay-based approach leads to a slightly worse workload distribution among the cache servers. However, the drop in the dev measure is relatively low (approximately 5 %) percent compared to the gain in the hit-rate (up to 40 %).

4.2.3 Observations on Scalability

Our specific evaluations of the Delaunay and the particle spring approach focus on the load balancing characteristics. For this reason, those experiments base on a constant set of resources (cache servers) and a constant amount of total workload. Only the data access patterns vary, for instance, through a moving hotspot simulation of data access. The experiments confirm that the general approach of load balancing works as long as the total amount of workload does not exceed the cumulated processing capabilities of the computing nodes participating in a distributed spatial cache.

Whenever the total workload exceeds the cumulated processing capabilities, load balancing is of little use. In this case, the resources have to be increased in order to match the increasing workload. One possibility to increase the resources in our distributed spatial cache is to add new cache servers, an action usually referred as *horizontal scale-out*.

In general, some of the added resources during a scale-out are typically spent on communication, as the processing entities need to collaborate to process incoming workload. In our case the distance base request forwarding as well as the propagation of workload statistics within the distributed spatial cache contribute to the communication overhead. A key question in this context is whether the total communication costs increase proportionally with the number cache servers during scale-out. For non-scalable systems the communication costs increase disproportionately with the number of nodes, making effective scale-out impossible. The feasibility of such a horizontal scale-out in a distributed system is often used as a key indicator to determine its scalability. For this reason, we focus on the impacts of a scale-out on the distributed spatial cache in this section.

A student project examined the scale-out characteristics of the Delaunay approach in a real cluster environment. In the project, a prototype was implemented and several experiments were conducted to measure the performance during scale-out. The methodology and the results of these experiments can be found in [Gessler et al., 2014]. These results can be directly transferred to the particle-spring approach, as similar communication overhead is expected (both approaches use distance based forwarding and work-

load propagation). In the following, we discuss the most significant parts of these experiments and refer the reader to the respective publication for further details.

The experiments were conducted in a cluster environment comprising up to 32 (virtual) computing nodes by executing the following process:

1. Deploy an initial set of cache servers ($n = 8$, $n = 12$, $n = 16$) in the cluster environment.
2. Initialize the cache focus positions of the initial cache servers according to a uniform distribution.
3. Triangulate the initial cache focus positions to establish the Delaunay-based link topology.
4. Initialize the benchmark client by choosing a data access pattern (Uniform, Hotspot). The uniform pattern results in uniformly distributed access in the the geographic data space whereas the hotspot pattern resembles a Gaussian distribution with a standard deviation of 18% of total data space area.
5. Start a benchmark client which sends 20 query requests per second to each cache server of the initial set.
6. Run the distributed spatial cache by executing the following two steps in parallel:
    a) Use distance based forwarding to find an adequate process server. Process the query on this server and send back the response to the benchmark client.
    b) Execute the scale-out mechanism of Section 4.2.2.4 and disable load balancing in order not to distort the measurements. The scale-out mechanism detects a local overload situation, whenever the workload of a cache server exceeds a certain threshold. The mechanism aims to reduce such a local overload situation by deploying a new cache server and by setting its cache focus nearby the overloaded server.
7. Measure the request-response time of each request with the benchmark client.

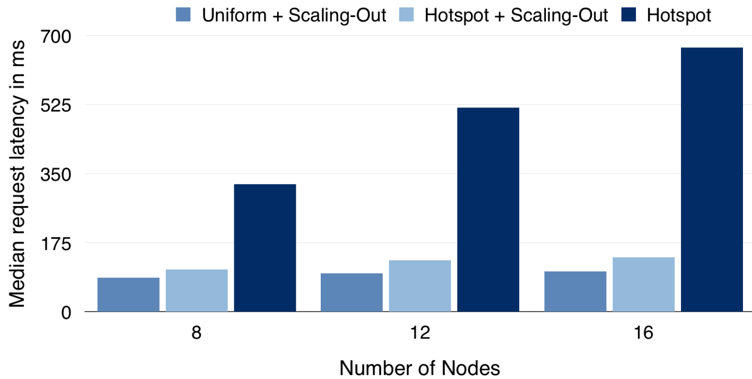


Figure 4.25: Median request latency in ms [Gessler et al., 2014]

Figure 4.25 shows the request latencies during the experiments. When the scale-out mechanism is disabled (experiment "Hotspot") the request latencies are worse by a magnitude compared to enabled scale-out (experiment "Uniform + Scaling-Out" and "Hotspot + Scaling-Out"). It can also be observed that the request latencies just slightly increase for increasing computing node numbers when scale-out is enabled.

These results show that the data access pattern aware scale-out mechanism can relieve local overload situations. Moreover, the results confirm that the scale-out process does not significantly increase the total communication costs in the distributed spatial cache. Thus, massive scale-out is feasible with our proposed system architecture.

4.3 Discussion

Based on our experiences with these concrete implementations and based on the results of our extensive evaluations, we assess the different cluster layer implementations in Table 4.2 with respect to the following two criteria:

- **Adaption effort:** This criterion measures the effort required to react to changed load situations.
- **Support of selective data access:** This criterion determines the performance in case of selective data access patterns.

In the table, we grade the cluster layer implementation with + for positive performance with respect to the examined criterion and – for negative performance.

The adaption effort is highest for the CAN-based implementation. This is due to the high costs of repartitioning the parcel-based layout. During repartitioning, a cache server must possibly hand over data to another cache server. In addition, even minor changes of the partitioning layout must be reflected in the overlay structure which drives additional effort. For the particle-spring and the Delaunay-based implementation, adaptation can be carried out by moving cache foci of the cache servers. This reduces the complexity of the adaptation operation, as the cache content of cache servers are automatically updated through the focused caching replacement strategy.

Our evaluation shows that the Delaunay-based system performs well, even under selective data access. This is due to the flexibility offered by the mathematical model of Delaunay-Triangulations. A Delaunay-Triangulation can effectively approximate any geometric feature. This is advantageous for adequate treatment of selective data access patterns. The particle-spring approach performs suboptimal in case of selective data access, as it is restricted by the physical properties of the particle-spring system. Likewise, the CAN-based approach is restricted by the naive split operation, which splits parcels in two equal parts. This leads to suboptimal distribution of workload among the cache servers for selective data access patterns.

4.4 Summary and Outlook

This chapter describes implementation concepts for the cluster layer of the distributed spatial cache. The major challenge in this layer is to find an adequate content arrangement among the cache servers. The content ar-

	Adaption Effort	Support of Selective Data Access
CAN-based	–	–
Particle-Spring	+	–
Delaunay	+	+

Table 4.2: The main characteristics of the cluster layer implementations

rangement affects the performance of data retrieval and the load distribution among the cache servers when the data is accessed. Therefore, an adequate content arrangement is a vital prerequisite for effective caching in a distributed system. In this chapter, we present two general principles for content arrangement. The first principle (see Section 4.1), separates the data space in partitions and assigns these partitions to the cache servers. The second principle (see Section 4.2), uses the specific cache replacement strategy focused caching to arrange the content among the cache servers.

Based on the two principles for content arrangement, we introduce three concrete implementations for the cluster layer. Our extended CAN-based implementation (cf. Section 4.1) uses spatial partitions for content arrangement. The particle-spring-based implementation (cf. Section 4.2.1) and the Delaunay-based (cf. Section 4.2.2) use focused caching for content arrangement.

We have evaluated all three cluster layer implementations through various experiments and have showed the general feasibility of each approach in concrete scenarios of our target application field.

With the discussion of the cluster layer in this chapter and the presentation of the cache layer in the previous chapter, we have covered the main technical challenges of our distributed spatial caching architecture. In the following chapter, we present various tools that support the operation of a distributed spatial cache in a cluster environment.

CHAPTER 5

Assessment Framework

The preceding chapters of this thesis present a motivation, discuss possible designs and propose implementations for our distributed spatial cache running in a cluster environment. In practice, it is a non-trivial task to set up, calibrate and maintain such a distributed system in a computer cluster for the following reasons:

- The cluster operator needs real applications to examine the correct functioning of the distributed spatial cache. Moreover, real applications help the operator to understand the substantial characteristics of the target application field, such as the data access behavior of clients.
- Our contributions include application-defined parameters, such as the stiffness parameter of the particle-spring approach (cf. Section 4.2.1). A cluster operator needs to determine appropriate parameter settings

in order to meet the specific requirements of the concrete application field.

- To determine suitable network configuration within the computer cluster executing the distributed spatial cache, the cluster operator needs to acquire a deep understanding for the network traffic expected in specific application scenarios.

This chapter provides various contributions which support the operator in respect to the above-mentioned challenges. In the following, Section 5.1 introduces a framework supporting rapid creation of visual applications with location-based context. With this framework the operator can quickly create real applications that help understanding specific data access behaviors and to examine the correct functioning of the distributed spatial cache. Section 5.2 introduces simulation tools for the distributed spatial cache. Using this simulation framework the operator can determine appropriate parameter settings for specific applications before the actual deployment. Finally, Section 5.3 introduces an evaluation prototype for cluster environments. With this tool the operator can deploy the distributed spatial cache in a computer cluster, generate the network traffic produced by typical spatial applications and inspect performance relevant parameters such as the latency induced by message propagation.

5.1 A Framework for Visual Example Applications

In this section, we describe the main features of NexusVIS, a visualization framework for location-based applications. Our description is based upon a previous work [Lübbe et al., 2010] to which we refer the reader for further details.

In general, any visualization aims at transforming complex or abstract data representations into perceivable visual representations which can be intuitively understood by the user. For this purpose, the visualization process can be divided into subtasks according to [Brodlie et al., 2004]. Inspired by this general approach, we identified four conceptual visualization process steps as part of our development efforts for NexusVIS: *select, map,*

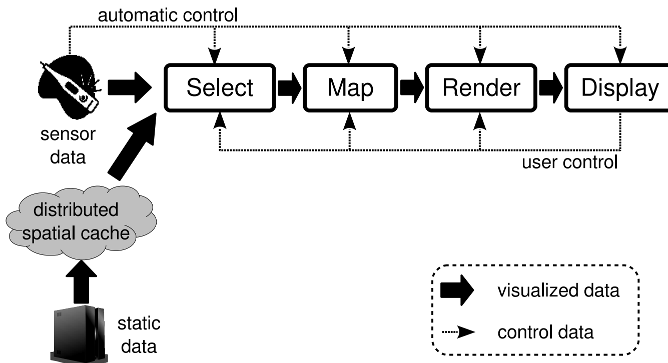


Figure 5.1: A visualization process using our distributed spatial cache

render and *display* (see Figure 5.1). The *select* step extracts relevant data, i.e. either live sensor information or rarely changing static data, such as map data. A distributed spatial cache keeps the most frequently requested static data for efficient access. From the extracted data, the *map* step creates abstract visual representations, i.e., a collection of geometric primitives such as points, lines, triangles, polygons or even text labels. The *render* step transforms them into a pixel-based image format. Finally, the *display* step presents the resulting stream of images to the user. The user can control the visualization using direct feedback links to each step. In addition, control data can also flow over forward links to each step to automatically control the visualization according to some sensor information, e.g. to automatically update the view to the current GPS position of a moving object.

NexusVIS bases upon NexusDS, an extensible middleware for distributed stream processing [Cipriani et al., 2010]. In this system processing semantics are encapsulated in *operators* which possess an arbitrary number of inputs and outputs. An operator without inputs, we denote as *source* and one without outputs as *sink*, respectively. We implemented the conceptual steps of the visualization process of Figure 5.1 as operators. By connecting the outputs and inputs of different operators, an application is able to

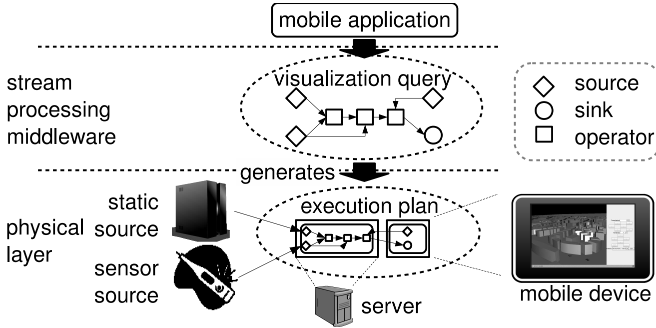


Figure 5.2: Execution of a visualization query [Lübbe et al., 2010]

define more complex query semantics. In this way, location-based applications express their visualization demands as visualization queries using the predefined visualization semantics of NexusVIS operators. As depicted in Figure 5.2, a location-based application sends a query to the stream processing middleware which distributes the operators to different processing nodes. As soon as the sources push data through the processing pipeline, the processing starts.

An exemplary query which a location-based application could express with NexusVIS is the visualization of a city map sector surrounding the current position of a mobile object. In this case, the viewport of the visualization has to be adapted to the current position of the mobile object, each time a position update is received. This causes frequent requests to fetch data related to the current vicinity of the moving object. For succeeding requests, we expect considerably high overlap, especially when the position of the mobile object has changed only marginally. For such scenarios, we consider a distributed spatial cache providing access to the most demanded data as beneficial.

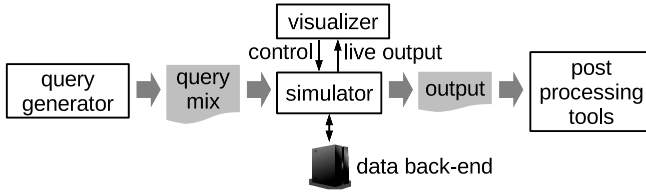


Figure 5.3: Simulation process [Lübbe and Cipriani, 2012]

5.2 A Simulation Framework

In this section, we present a simulation framework for the distributed spatial cache. Our description in this section bases upon a previous work [Lübbe and Cipriani, 2012].

The simulation framework enables the operator to determine settings for relevant system parameters prior to the execution. For this purpose, our framework provides the following features:

- **Emulation** of geo-referenced query workloads according to different mobility models.
- **Live observance** of the current cache performance, such as hit-rate and workload distribution.
- **Live calibration** of system parameters during simulation via a graphical user interface.
- **Live visualization** of the logical link overlay, the current data access patterns, the current cache content and more.

Figure 5.3 depicts the general simulation process with our framework. In the following, we briefly describe all relevant components which are involved in the simulation process.

The **query generator** component creates a file emulating the workload of all clients. The workload emulation consists of mobile users which request data in the vicinity of their current position using range queries. The query generator can distribute the target regions of the range queries according

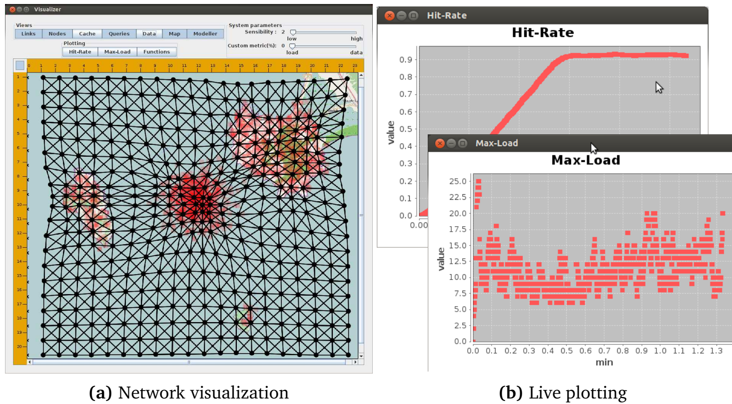
to different user mobility models. The simplest model generates a *uniform distribution* of target regions within a user-defined area. As this data access pattern possesses hardly any locality, the operator can use it as a benchmark for the worst case scenario in respect to caching. Moreover, the target regions can be distributed according to a *Gaussian distribution* emulating an access hotspot at a certain geographic position. Finally, the *moving Gaussian distribution* represents an access pattern possibly induced by a traffic jam moving in space and time.

The **simulator** component performs a discrete event-driven simulation of all cache servers within the distributed spatial cache during request processing. We implemented the simulator using PeerSim [Montresor and Jelasity, 2009], a Java-based simulation framework for peer-to-peer networks. In this framework, we included our implementations for the cache layer and the cluster layer (cf. Chapter 3 and Chapter 4) in order reproduce the behavior of real cache servers.

During request processing, the simulated cache servers request missing data from the **data back-end**. The back-end uses geographic data extracted from OpenStreetMap [OpenStreetMap, 2011]. As the open source community around OpenStreetMap provides a wide range of tools for export and import to and from various data formats, other data sources can be included easily.

The **visualizer** component provides a graphical interface for the simulation, as depicted in Figure 5.4. It enables the cluster operator to introspect various relevant aspects of the run-time behavior of the distributed spatial cache, such as the logical link structure between cache nodes (cf. Figure 5.4a) and performance relevant indicators via graph plotting (cf. Figure 5.4b). Through a control panel interface the operator can change simulation parameters and instantly observe the effects in the simulation.

In addition to the live output of the visualization provided by the graphical interface, the simulation results are written to an output file using the widely supported comma separated value format (CSV). For further analysis of the output, our framework provides a set of **post processing tools** for



(a) Network visualization

(b) Live plotting

Figure 5.4: Graphical interface of the simulation

interpolation, aggregation and more. All tools conform to the CSV format so that alternative post processing tools or plotting tools can be applied.

We used the simulation framework to evaluate our approaches for the cluster layer (cf. Section 4.1.1.6, 4.2.1.7 and 4.2.3). For this purpose, we emulated several data access patterns of clients to generate a benchmark workload. We used the live observance capabilities of the framework to determine the parameters for our experiments, such as the number of simulated cache servers, the cache capacity and others. Then, we ran our experiments using the framework's simulator engine. We used the charting tool Gnuplot [Williams et al., 2010] to visualize the generated simulation results.

Further details about the simulation framework can be found in our publication [Lübbe and Cipriani, 2012]. We continue with describing a framework for execution in a real cluster environment in the following section.

5.3 An Evaluation Framework for Cluster Environments

In a student project accompanying this work, a framework was implemented to evaluate the distributed spatial cache in a real cluster environment. The framework was used to evaluate scalability characteristics of the distributed spatial cache (cf. Section 4.2.3). In this section, we describe the main characteristics of this framework and refer the reader to [Gessler et al., 2014] for more details.

The general architecture is sketched in Figure 5.5. The evaluation framework uses the following process to conduct measurements in the cluster environment:

1. The topology creator is used to generate the link structure of the distributed spatial cache.
2. The distributed spatial cache is deployed on the computing cluster by installing and starting the cache server software on the computing nodes. The framework then establishes the generated link structure between the nodes.
3. The benchmark client sends requests to each of the deployed cache servers with a configurable request rate.
4. The cache servers use distance-based forwarding to find an adequate server for processing, process the request and send the response back to the benchmark client.
5. The benchmark client measures and logs the request-response time of each request.

With the cluster evaluation framework, the operator can measure the overall latencies which are expected for handling client requests. The expected latencies depend on various issues, such as the current layout of the logical link structure, the size of the distributed spatial cache, the current network congestion and so on. The cluster evaluation framework helps the cluster operator to disclose possible performance bottlenecks within the cluster environment.

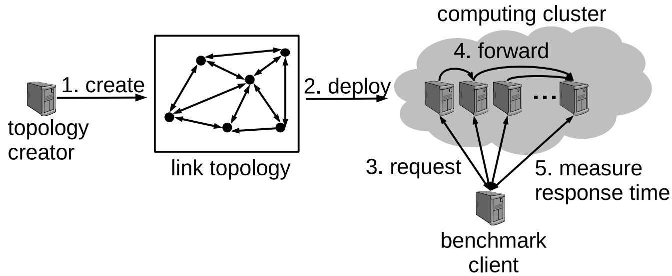


Figure 5.5: Evaluation prototype for cluster environments

5.4 Summary and Outlook

This chapter describes, three frameworks the operator can use to assess operation mode of the distributed spatial cache. The first framework allows to create real location-based applications (cf. Section 5.1), the second provides a simulation with live-visualization of a distributed spatial cache (cf. Section 5.2) and the third enables the execution in a real cluster environment (cf. Section 5.3).

The following chapter concludes this work and provides an overview on ongoing research topics.

CHAPTER 6

Conclusion

In this chapter, we review our work in Section 6.1, discuss its practical use in Section 6.2 and finally present our ideas about future research topics in Section 6.3.

6.1 Review

High availability has become an important issue for the provisioning of spatial data. As the consumers of spatial data often request similar things (e.g., hotels close to the main station), the requests overlap to a great extent. To exploit this high data access locality, we propose a separate caching middleware in between the data provider and the data consumer which stores the most frequently used data. For this purpose, we devised key concepts for caching spatial data on a single cache server. To achieve high throughput

with an increasing number of clients, we combine multiple cache servers into a peer-to-peer-based network structure that we denote as distributed spatial cache. Within this distributed spatial cache availability is achieved trough:

- automatic **scaling** mechanisms which bring the currently offered processing capacity in line with the current demand,
- advanced **load-balancing** processes which take care of distributing the workload between the cache server
- a dedicated **caching strategy** which considers the data access patterns of location-based applications.

6.2 Practical Use

The focus of this work is on presenting the general concepts and methods which are necessary to build a distributed spatial cache. In addition to that, we proposed three different imnplementations of the distributed spatial cache to show the practicability of our ideas (see Section 4). Various evaluations of these implementations under realistic conditions underline the practicability of our approach. On top of that, we developed tools that help a cluster operator to run such a distributed spatial cache (see Section 5).

The recent trend is to use a network of remote servers to provide services to the end user, rather than a personal computers. This trend is currently often summarized under the unspecific term of *cloud computing*. Gained scalability is one of the main reasons for cloud computing. In this way this work follows the trend, as the distributed spatial cache is particularly suited to be hosted over the Internet. Beyond that, its architecture facilitates the separation of the concerns, as the cache layer is used to gain scalability. This helps to disburden providers of spatial data and decrease the overall costs of data provisioning.

6.3 Future Prospects

Beyond the contributions of this work, several further developments are possible. In the following, we briefly describe some possible extensions that can be incorporated into our work:

- **Multi target optimization:** For load balancing in our distributed spatial cache, we identified two major indicators: data skew and workload. In general, these two optimization goals are independent, as an optimal workload distribution does not necessarily imply an even distribution of cache content between the cache servers and vice versa. In this work, we let the application relate the aspects using fused optimization functions. The advantage of this approach is that the application can tune the optimization mechanism towards its needs. But what if the application is not able to define such a fused optimization criterion? In such a case, multi target optimization techniques could possibly support the application. It still has to be investigated whether these methods are efficient enough to achieve a high overall performance of the distributed spatial cache.
- **Multiple dimensions:** For many location-based applications two dimension suffice to define positions, as these can be easily transformed to geographic coordinates using the respective map transformation methods. However, in some cases additional dimensions are required to allocate data. Indoors, for instance, an indication of the floor is often crucial for selecting relevant information. Even though our current implementations of the distributed spatial cache focuses on two dimensions, many of the used methods can be generalized to more dimensions in principle. Particularly interesting is the question how to define overlay topologies in the multi-dimensional case with consideration of constraints such as physical stability and routing efficiency.
- **Multiple tenants:** As of today, many popular location-based applications co-exist. Some of them share a considerable overlap in the data they require. Thus, a single consolidated distributed cache which serves many applications at the same time might be the most efficient setup in this situation. For such a multi tenant cache, additional mea-

sures have to be taken. For instance, it must be ensured that caching resources are fairly distributed among tenants during the resource allocation of the load-balancing process. Moreover, if the applications incorporate private data which is worth protecting then additional security mechanisms are required in order to isolate the tenants.

List of Figures

1.1 System overview 23

2.1 System Environment 31
2.2 A cache server's internal design 36
2.3 Complex back-end part of a spatial query 40
2.4 Design variants for the distributed spatial cache 50
2.5 Proposed design of the cluster edge. 53
2.6 Proposed software architecture 57

3.1 Spatial partitions as caching units 61
3.2 Storage modes for the partition containers 62
3.3 Query processing using *focused caching*. 64
3.4 Space overhead for caching of a concrete application scenario 66
3.5 Number of operations necessary to process a request 68
3.6 Data identification accuracy 69
3.7 Back-end request for a square query region 70

3.8 Query processing in the type extended parcel-based cache. . 72

4.1 A geographically partitioned distributed spatial cache 78
4.2 Adaptive partitioning 82
4.3 Results of the simulation Part 1 87
4.4 Results of the simulation Part 2 89
4.5 Content arrangement based on focused caching 91
4.6 Data access in the distributed spatial cache 93
4.7 Load balancing through spring contraction. 94
4.8 Particle topology 98
4.9 Iterative spring relaxation into low energy state. 100
4.10 Scale-out of the Distributed Spatial Cache 103
4.11 Replacement process of failed cache servers. 104
4.12 Visualization of the distributed spatial cache 107
4.13 Impacts of gravity scale parameter β 108
4.14 Impacts of the load weight parameter α 109
4.15 A Delaunay-based link topology with circumcircles 112
4.16 Move a node's cache focus position 115
4.17 From the actual to the demanded distribution of resources . 116
4.18 Geographic distribution ρ_G 119
4.19 Data skew ρ_N 120
4.20 Dynamic load distribution ρ_D 122
4.21 Hybrid load distribution ρ_H 123
4.22 Data access pattern during simulation 126
4.23 Illustration of the adaption process 127
4.24 Performance of the Delaunay-based cluster layer 129
4.25 Median request latency in ms [Gessler et al., 2014] 132

5.1 A visualization process using our distributed spatial cache . . 137
5.2 Execution of a visualization query [Lübbe et al., 2010] 138
5.3 Simulation process [Lübbe and Cipriani, 2012] 139
5.4 Graphical interface of the simulation 141
5.5 Evaluation prototype for cluster environments 143

List of Tables

1.1 Support level of technologies for specific requirements. 22

2.1 Comparison of cache granularities. 42
2.2 Properties of Cache Memory Types. 47

4.1 Routing tables 99
4.2 The main characteristics of the cluster layer implementations 134

List of Listings

4.1 Iterative spring relaxation algorithm 101
4.2 Algorithm for a ρ-distributed topology of cache foci 117

Bibliography

[Aberer et al., 2005] Aberer, K., Datta, A., Hauswirth, M., and Schmidt, R. (2005). Indexing data-oriented overlay networks. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 685–696. VLDB Endowment.

[Bhide, 1988] Bhide, A. (1988). An analysis of three transaction processing architectures. In *Proceedings of the 14th International Conference on Very Large Data Bases*, VLDB '88, pages 339–350, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Boboila and Desnoyers, 2010] Boboila, S. and Desnoyers, P. (2010). Write endurance in flash drives: Measurements and analysis. In *FAST*, volume 10, pages 9–9.

[Bose et al., 1999] Bose, P. et al. (1999). Online routing in triangulations. In *ISAAC*, pages 113–122, London, UK. Springer-Verlag.

[Brisco, 1995] Brisco, T. (1995). Rfc 1794: Dns support for load balancing. RFC Editor , United States.

[Brodlie et al., 2004] Brodlie et al., K. (2004). Visualization in grid computing environments. In *VIS '04*. IEEE CS Press.

[Carey et al., 1994] Carey, M. J., Franklin, M. J., and Zaharioudakis, M. (1994). Fine-grained sharing in a page server oodbms. pages 359–370.

[Cattell, 2011] Cattell, R. (2011). Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27.

[Chang et al., 2008] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26.

[Chaudhuri, 1998] Chaudhuri, S. (1998). An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 34–43, New York, NY, USA. ACM.

[Chaudhuri, 2007] Chaudhuri, S. (2007). Self-tuning database systems: A decade of progress. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB)*, pages 3–14. VLDB Endowment.

[Chrisman, 1991] Chrisman, N. R. (1991). *Geographical information systems: principles and applications.*, chapter The error component in spatial data, page 165–174. Harlow, Longman/New York, John Wiley & Sons Inc.

[Cipriani et al., 2010] Cipriani, N., Lübbe, C., and Moosbrugger, A. (2010). Exploiting Constraints to Build a Flexible and Extensible Data Stream Processing Middleware. In *The Third International Workshop on Scalable Stream Processing Systems*, pages 1–8. IEEE Computer Society.

[Cole, 2013] Cole, D. (2013). iphone map app directs fairbanks drivers onto airport taxiway. accessed on Dec. 16th 2013.

[Costa et al., 2013] Costa, P., Donnelly, A., O'Shea, G., and Rowstron, A. (2013). Camcubeos: A key-based network stack for 3d torus cluster topologies. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 73–84, New York, NY, USA. ACM.

[Dabek et al., 2004] Dabek, F., Cox, R., Kaashoek, F., and Morris, R. (2004). Vivaldi: A decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26.

[Dar et al., 1996] Dar, S., Franklin, M. J., Jónsson, B. T., Srivastava, D., and Tan, M. (1996). Semantic data caching and replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 330–341, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[de Berg et al., 2000] de Berg, M. et al. (2000). *Computational Geometry: Algorithms and Applications*. Springer.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

[DeWitt and Gray, 1992] DeWitt, D. and Gray, J. (1992). Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98.

[DeWitt et al., 1990] DeWitt, D. J., Futtersack, P., Maier, D., and Velez, F. (1990). A study of three alternative workstation server architectures for object-oriented database systems. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 107–121, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Dinger and Waldhorst, 2009] Dinger, J. and Waldhorst, O. (2009). Decentralized bootstrapping of p2p systems: A practical view. In Fratta, L., Schulzrinne, H., Takahashi, Y., and Spaniol, O., editors, *NETWORKING 2009*, volume 5550 of *Lecture Notes in Computer Science*, pages 703–715. Springer Berlin Heidelberg.

[Du and Gunzburger, 2002] Du, Q. and Gunzburger, M. (2002). Grid generation and optimization based on centroidal voronoi tessellations. *Appl. Math. Comput.*, 133(2-3):591–607.

[Effelsberg and Haerder, 1984] Effelsberg, W. and Haerder, T. (1984). Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595.

[Foursquare, 2013] Foursquare (2013). Foursquare. accessed on Dec. 16th 2013.

[Gessler et al., 2014] Gessler, A., Hanna, S., and Smith, A. M. (2014). Scaling in a distributed spatial cache overlay. In *Proceedings "Informatiktage 2014"*. Gesellschaft für Informatik e.V. (GI).

[Gilbert and Lynch, 2002] Gilbert, S. and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59.

[Grochowski and Hoyt, 1996] Grochowski, E. and Hoyt, R. F. (1996). Future trends in hard disk drives. *Magnetics, IEEE Transactions on*, 32(3):1850–1854.

[Grossmann et al., 2009] Grossmann, M., Hönle, N., Lübbe, C., and Weinschrott, H. (2009). An abstract processing model for the quality of context data. In *Proceedings of the 1st international conference on Quality of context*, QuaCon'09, pages 132–143, Berlin, Heidelberg. Springer-Verlag.

[Güting, 1994] Güting, R. H. (1994). An introduction to spatial database systems. *The VLDB Journal*, 3(4):357–399.

[Guttman, 1984] Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA. ACM.

[Härder and Bühmann, 2008] Härder, T. and Bühmann, A. (2008). Value complete, column complete, predicate complete. *The VLDB Journal*, 17(4):805–826.

[Hu et al., 2005] Hu, H., Xu, J., Wong, W. S., Zheng, B., Lee, D. L., and Lee, W.-C. (2005). Proactive caching for spatial queries in mobile environments. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 403–414.

[Hudlet and Schall, 2011] Hudlet, V. and Schall, D. (2011). Ssd != ssd - an empirical study to identify common properties and type-specific behavior. In Härder, T., Lehner, W., Mitschang, B., Schöning, H., and Schwarz, H., editors, *BTW*, volume 180 of *LNI*, pages 430–441. GI.

[Hunter et al., 2009] Hunter, G., Bregt, A., Heuvelink, G., Bruin, S., and Virrantaus, K. (2009). Spatial data quality: Problems and prospects. In Navratil, G., editor, *Research Trends in Geographic Information Science*, Lecture Notes in Geoinformation and Cartography, pages 101–121. Springer Berlin Heidelberg.

[Kim and Feamster, 2013] Kim, H. and Feamster, N. (2013). Improving network management with software defined networking. *Communications Magazine, IEEE*, 51(2):114–119.

[Kim et al., 1989] Kim, W., Ballou, N., Chou, H.-T., Garza, J. F., and Woelk, D. (1989). Object-oriented concepts, databases, and applications. chapter Features of the ORION Object-oriented Database System, pages 251–282. ACM, New York, NY, USA.

[Kiyoo Itoh et al., 1995] Kiyoo Itoh, B., Sasaki, K., and Nakagome, Y. (1995). Trends in low-power ram circuit technologies. *Proceedings of the IEEE*, 83(4):524–543.

[Konstantinou et al., 2011] Konstantinou, I., Angelou, E., Boumpouka, C., Tsoumakos, D., and Koziris, N. (2011). On the elasticity of nosql databases over cloud management platforms. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 2385–2388, New York, NY, USA. ACM.

[Krishnamurthy and Wills, 1997] Krishnamurthy, B. and Wills, C. E. (1997). Study of piggyback cache validation for proxy caches in the world wide web. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, USITS'97, pages 1–1, Berkeley, CA, USA. USENIX Association.

[Kubach and Rothermel, 2001] Kubach, U. and Rothermel, K. (2001). Exploiting location information for infostation-based hoarding. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, MobiCom '01, pages 15–27, New York, NY, USA. ACM.

[Lawson, 1977] Lawson, C. L. (1977). Mathematical Software III; Software for C1 surface interpolation. pages 161–194. Academic Press, New York.

[Lee and Lam, 2008] Lee, D.-Y. and Lam, S. (2008). Efficient and accurate protocols for distributed delaunay triangulation under churn. In *Network Protocols, 2008. ICNP 2008. IEEE International Conference on*, pages 124–136.

[Leiserson, 1985] Leiserson, C. E. (1985). Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901.

[Li, 2008] Li, J. (2008). On peer-to-peer (p2p) content delivery. *Peer-to-Peer Networking and Applications*, 1(1):45–63.

[Liu and Cao, 1998] Liu, C. and Cao, P. (1998). Maintaining strong cache consistency in the world-wide web. In *In Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, pages 445–457.

[Lübbe et al., 2011] Lübbe, C., Brodt, A., Cipriani, N., Großmann, M., and Mitschang, B. (2011). Disco: A distributed semantic cache overlay for location-based services. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, volume 1, pages 17–26.

[Lübbe et al., 2010] Lübbe, C., Brodt, A., Cipriani, N., and Sanftmann, H. (2010). Nexusvis: A distributed visualization toolkit for mobile applications. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, pages 841–843.

[Lübbe and Cipriani, 2012] Lübbe, C. and Cipriani, N. (2012). Simpl: A simulation platform for elastic load-balancing in a distributed spatial cache overlay. In *Mobile Data Management (MDM), 2012 IEEE 13th International Conference on*, pages 340–343.

[Lübbe and Mitschang, 2013a] Lübbe, C. and Mitschang, B. (2013a). An elastic cache infrastructure through multi-level load-balancing. In Helfert, M., Francalanci, C., and Filipe, J., editors, *DATA*, pages 183–190. SciTePress.

[Lübbe and Mitschang, 2013b] Lübbe, C. and Mitschang, B. (2013b). Holistic load-balancing in a distributed spatial cache. In *Mobile Data Management (MDM), 2013 IEEE 14th International Conference on*, vol-

ume 1, pages 267–270.

[Lübbe et al., 2012] Lübbe, C., Reuter, A., and Mitschang, B. (2012). Elastic load-balancing in a distributed spatial cache overlay. In *Mobile Data Management (MDM), 2012 IEEE 13th International Conference on*, pages 11–20.

[Macqueen, 1967] Macqueen, J. B. (1967). Some methods of classification and analysis of multivariate observations. In *Berkeley Symp. on Mathematical Statistics and Probability*, pages 281–297.

[Montresor and Jelasity, 2009] Montresor, A. and Jelasity, M. (2009). PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA.

[Nicklas et al., 2004] Nicklas et al., D. (2004). On building location aware applications using an open platform based on the NEXUS augmented world model. *Software and System Modeling*, 3(4).

[Nishtala et al., 2013] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. (2013). Scaling memcache at facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA. USENIX Association.

[Open Geospatial Consortium Inc., 2011] Open Geospatial Consortium Inc. (2011). Opengis implementation standard for geographic information - simple feature access - part 1: Common architecture.

[OpenStreetMap, 2011] OpenStreetMap (2011). `http://www.openstreetmap.org`, accessed on Nov. 2011.

[Park et al., 1999] Park, H.-H., Lee, C.-G., Lee, Y.-J., and Chung, C.-W. (1999). Early separation of filter and refinement steps in spatial query optimization. In *Database Systems for Advanced Applications, 1999. Proceedings., 6th International Conference on*, pages 161–168.

[Petroski, 1996] Petroski, H. (1996). *Invention by Design: How Engineers Get from Thought to Thing*. Harvard University Press.

[Pietzuch et al., 2006] Pietzuch, P. R., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., and Seltzer, M. I. (2006). Network-Aware Operator Placement for Stream-Processing Systems. In *International Conference on Data Engineering*.

[Plugge et al., 2010] Plugge, E., Mebrey, P., and Hawkins, T. (2010). *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Paul Manning.

[Pritchett, 2008] Pritchett, D. (2008). Base: An acid alternative. *Queue*, 6(3):48–55.

[Pröll et al., 2013] Pröll, S., Zangerle, E., and Gassler, W. (2013). *MySQL 5.6: das umfassende Handbuch*. Galileo Computing : Datenbanken. Galileo Press.

[Rahm, 1993] Rahm, E. (1993). Evaluation of closely coupled systems for high performance database processing. In *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 301–310.

[Ratnasamy et al., 2001] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. (2001). A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172.

[Rowstron and Druschel, 2001] Rowstron, A. I. T. and Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK. Springer-Verlag.

[Schwarz et al., 2004] Schwarz, T., Iofcea, M., Grossmann, M., Hönle, N., Nicklas, D., and Mitschang, B. (2004). On efficiently processing nearest neighbor queries in a loosely coupled set of data sources. In *Proceedings of the 12th annual ACM international workshop on Geographic information systems*, GIS '04, pages 184–193, New York, NY, USA. ACM.

[Smith, 1978] Smith, A. J. (1978). Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.*, 3(3):223–247.

[Stepanov et al., 2003] Stepanov et al., I. (2003). A meta-model and framework for user mobility in mobile networks. In *ICON*, pages 231 – 238.

[Stoica et al., 2001] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160.

[Stonebraker, 1986] Stonebraker, M. (1986). The case for shared nothing. *Database Engineering*, 9:4–9.

[Stonebraker et al., 2005] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., and Zdonik, S. (2005). C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment.

[Stonebraker et al., 1976] Stonebraker, M., Held, G., Wong, E., and Kreps, P. (1976). The design and implementation of ingres. *ACM Trans. Database Syst.*, 1(3):189–222.

[Stonebraker and Moore, 1995] Stonebraker, M. and Moore, D. (1995). *Object Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Urgaonkar et al., 2001] Urgaonkar, B., Ninan, A., Raunak, M., Shenoy, P., and Ramamritham, K. (2001). Maintaining mutual consistency for cached web objects. In *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 371–380.

[Wang et al., 2005] Wang et al., H. (2005). Aspen: an adaptive spatial peer-to-peer network. In *ACM GIS*, pages 230–239, NY, USA. ACM.

[Williams et al., 2010] Williams, T., Kelley, C., and many others (2010). Gnuplot 4.4: an interactive plotting program. `http://gnuplot.sourceforge.net/`.

[Wood, 2008] Wood, J. (2008). Filter and refine strategy. In Shekhar, S. and Xiong, H., editors, *Encyclopedia of GIS*, pages 320–320, Boston, MA. Springer US.

[Yin et al., 1999] Yin, J., Alvisi, L., Dahlin, M., and Lin, C. (1999). Volume leases for consistency in large-scale systems. *Knowledge and Data Engineering, IEEE Transactions on*, 11(4):563–576.

[Yoo, 2007] Yoo, J. S. (2007). *Spatial Query Processing and Data Mining Methods for Location Based Services*. ProQuest Information and Learning Company,Ann Arbor, United States.

[Zhang et al., 2007] Zhang, J., Zhang, G., and Liu, L. (2007). Geogrid: A scalable location service network. In *Proceedings of 27th IEEE International Conference on Distributed Computing Systems (ICDCS)*.

[Zhao et al., 2001] Zhao, B. Y., Kubiatowicz, J. D., and Joseph, A. D. (2001). Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, Berkeley, CA, USA.